Interleaved Multi-Vectorizing

Zhuhe Fang, Beilei Zheng, Chuliang Weng*

School of Data Science and Engineering, East China Normal University, China
{zhfang, zhengbeilei}@stu.ecnu.edu.cn, clweng@dase.ecnu.edu.cn

ABSTRACT

SIMD is an instruction set in mainstream processors, which provides the data level parallelism to accelerate the performance of applications. However, its advantages diminish when applications suffer from heavy cache misses. To eliminate cache misses in SIMD vectorization, we present interleaved multi-vectorizing (IMV) in this paper. It interleaves multiple execution instances of vectorized code to hide memory access latency with more computation. We also propose residual vectorized states to solve the control flow divergence in vectorization. IMV can make full use of the data parallelism in SIMD and the memory level parallelism through prefetching. It reduces cache misses, branch misses and computation overhead to significantly speed up the performance of pointer-chasing applications, and it can be applied to executing entire query pipelines. As experimental results show, IMV achieves up to 4.23X and 3.17X better performance compared with the pure scalar implementation and the pure SIMD vectorization, respectively.

PVLDB Reference Format:
Zhuhe Fang, Beilei Zheng, Chuliang Weng. Interleaved Multi-Vectorizing. *PVLDB*, 13(3): 226-238, 2019.
DOI: https://doi.org/10.14778/3368289.3368290

1. INTRODUCTION

SIMD (Single Instruction Multiple Data) is an instruction set available in modern processors. With SIMD, a single instruction is executed in parallel on multiple data points as opposed to executing multiple instructions. SIMD offers higher data parallelism with larger vectors, which today are up to 512 bits in the latest AVX512 instruction set. It is widely studied to speed up operations in databases, graphs and other domains, including join, partitioning, sorting, bloom filter, selection, and compression [28, 9, 29, 13, 14]. These operations benefit from the vectorized execution using SIMD to reduce computation overhead and branch misses.

However, gains from SIMD diminish when operations frequently access memory data, such as probing hash tables, probing bloom filters and searching trees [28, 17], because these operations are dominated by memory access latency, when they working on large datasets that cannot fit into the processor's cache. The memory access latency is not alleviated with SIMD, although SIMD issues more memory accesses in a vector. Due to the lockstep in SIMD, even if some requested data elements from a vector hit in cache, they still cannot continue until cache misses of the others in the vector are solved. Besides, the growing gap between CPU and memory speeds makes the bottleneck of memory access worse, while SIMD just accelerates the CPU processing instead of data accessing.

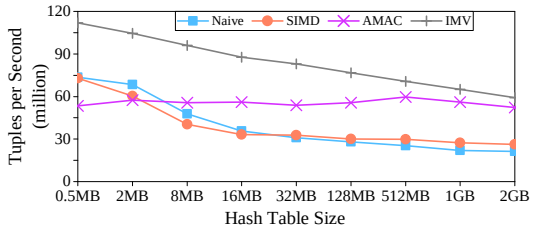


Figure 1: The performance of hash join probe

We demonstrate the impact of memory accesses on the hash join probe through an experiment, and we defer a description of the experimental setup (hardware configuration, workloads) to Section 5.1. The probe works on a chained hash table, which size ranges from 0.5 MB to 2 GB. The probing table contains 800 MB data. The two tables are generated with zipfian factor 1. We take the state-of-the-art full vectorization to vectorize probe on the chained hash table, and compare the vectorized probe with its scalar counterpart. In this experiment, we execute the probe operation using a single thread to eliminate the effects of multi-threading. As shown in Figure 1, the throughput of the vectorized probe using SIMD is comparable to that of the naive probe in scalar code with increasing hash table size. Their throughput significantly declines with increasing hash table size until the size approaches the capacity of memory cache (about 12 MB). Beyond that, their throughput slightly drops with increasing data, due to heavy cache misses.

In scalar code, such cache misses can be effectively reduced using software prefetching, i.e., manually inserting prefetching instructions to load data beforehand into cache. With prefetching, GP and SPP [6] are two ways to improve hash join performance, but they cannot solve irregular data accesses well, thus AMAC [20] is proposed. In fact, irregular data accesses are common in pointer-chasing applications, such as probing a hash table and traversing a tree or a graph. Therefore, we also take AMAC to speed up probe on a chained hash table. Figure 1 demonstrates AMAC can double the throughput of probe on a large hash table.

Prior work has not studied the effect of cache misses on SIMD operations. Specifically, a part of previous studies attempt to rearrange data layout to increase data locality, and then benefit from the hardware prefetching [14, 32, 18]. Other studies just preliminarily take the software prefetching. For example, SPP prefetches data while traversing trees. However, these paths are of uniform depth [18], and this method cannot be applied to irregular data ac-

*Chuliang Weng is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 13, No. 3
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3368289.3368290

cesses. Besides, the software prefetching is also adopted to speed up sequential data accesses in [7, 15], slightly reducing cache misses. Furthermore, ROF [26] links SIMD-optimized code and prefetching-optimized code in an entire query plan, instead of fully combining SIMD and prefetching, thus ROF improves performance to some extent, but could not benefit from SIMD and prefetching within an operator at the same time.

Since GP, SPP and AMAC successfully avoid cache misses in scalar code, combining them with SIMD vectorization is a straightforward way to solve cache misses in vectorized code. It is easy to vectorize each stage in prefetching-optimized techniques. However, such a method suffers from bubbles in vectors due to the control flow divergence in irregular data accesses. For example, when a vector of tuples probe multiple hash buckets in parallel, a part of which may terminate earlier due to fewer nodes in buckets, leaving their vector lanes idle in the following processing. To avoid the bubbles, it is better to apply the full vectorization [28] to the prefetching-optimized algorithms. This can entirely occupy vectors but not fully use vectors, because after filling the idle vector lanes with new tuples, the older tuples have to repeat some steps to keep pace with new ones, still wasting vector lanes. Even worse, such full vectorization cannot handle the control flow divergence from the general *if* and *loop* statements with more valid branches, because it cannot handle more branches within a vector.

Combining the prefetching-optimized techniques with SIMD vectorization can neither make full use of vectors nor be applied to general cases, wasting the data level parallelism and the memory level parallelism. In response, we propose interleaved multi-vectorizing (IMV), which runs multiple instances of vectorized code in an interleaved way. Once an instance encounters immediate memory accesses, it issues data prefetching and switches to other running instances, trying to overlap memory accesses with computation among these instances. We also introduce residual vectorized states to solve the control flow divergence. In a running instance, when vectors in a state become not full, such a divergent state will integrate with the residual state at that point, after which vectors in the state become full and continue to execute its next states, or become empty to restart. So the divergent state does not influence the following execution, and all instances could make full use of vector lanes. This can better solve the divergence in general *if* and *loop* statements. As illustrated in Figure 1, IMV significantly speeds up the throughput of the hash join probe, and it is faster than AMAC because of reducing branch misses.

The contributions of this paper are concluded as follows.

(1) We are the first to comprehensively study how to reduce cache misses in SIMD vectorization. Combining prefetching into SIMD vectorization could exploit data-level and memory-level parallelism simultaneously, not just for a single operator but also in a query pipeline.

(2) We directly or fully apply SIMD vectorization to accelerate prefetching-optimized techniques, and find those two ways neither solve the control flow divergence well, nor are generally applied to *if* and *loop* statements.

(3) We propose Interleaved Multi-Vectorizing (IMV) to perfectly combine SIMD vectorization and prefetching.

(4) We introduce the residual vectorized state to solve the control flow divergence well in IMV, which could be applied to general *if* and *loop* statements.

(5) We conduct comprehensive experiments on a CPU processor and a Phi co-processor. As the results show, IMV achieves up to 4.23X and 3.17X better performance compared with the pure scalar implementation and the pure vectorization, respectively.

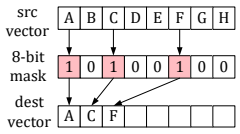


Figure 2: Compress

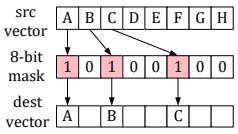


Figure 3: Expand

The remainder of this paper is organized as follows. Section 2 describes some related background, including SIMD vectorization and prefetching. Section 3 analyzes how to combine SIMD with previous prefetching-optimized techniques. Section 4 presents IMV and the residual vectorized state to solve the control flow divergence. Section 5 demonstrates our experimental evaluation. Section 6 discusses IMV in complex queries and IMV automation. Finally, we introduce some related work in Section 7 and conclude this paper in Section 8.

2. BACKGROUND

In this section we first briefly discuss advantages of SIMD, and introduce the instructions accelerating operations in databases. Then we focus on the issue of cache misses, and its potential solutions, including hardware and software prefetching. Furthermore, we analyze existing prefetching-based research efforts.

2.1 SIMD Vectorization

Modern (co-)processors provide SIMD instructions to operate multiple data elements using a single instruction. SIMD offers higher data level parallelism (DLP) with the larger vectors, which today are up to 512 bits in AVX512, the latest instruction set[1]. So a vector has 8 lanes to process 64-bit integers in AVX512. SIMD not only reduces computation overhead, but also can convert control flow to data flow to avoid branches [13] in some cases. For example, in a simple *if* statement, SIMD executes all branches then combines the results from branches instead of entering some branches according to conditions.

SIMD instructions can not only accelerate computation but also facilitate accessing data from memory and swizzle data elements among SIMD vectors. In detail, `gather` selectively reads data from non-contiguous memory addresses into a vector. The reversing operation can be implemented by `scatter`. In particular, a sub-collection of AVX512 called AVX512PF supports prefetching for `gather/scatter`. Additionally, `compress` packs active data elements (indicated by a mask) contiguously to a target vector. In contrast, `expand` stores the contiguous elements of a vector to some specific positions (hinted by a mask) of another vector. The latter two operations are illustrated in Figures 2 and 3.

The design principles are presented to fully vectorize main memory database operations in [28]. It defines that an algorithm is fully vectorized, only if its vectorized counterpart executes $O(f(n)/W)$ vector instructions instead of $O(f(n))$ scalar instructions, where W is the vector length. Actually the definition excludes random memory accesses, because executing W cache accesses per cycle is an impractical hardware design. As a result, the full vectorization is difficult to be achieved in memory-intensive operations.

2.2 Prefetching

The much growing gap between processor speed improvements and memory speed improvements results in the former's improvements being masked by the relatively slow improvements of the latter, known as "memory wall" [25][34]. Caching is an effective mechanism to mitigate the negative influence. However, cache

[1]https://software.intel.com/sites/landingpage/IntrinsicsGuide/

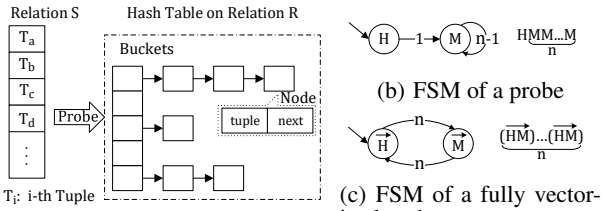
misses are time-consuming, almost costing 80ns - 200ns. The hardware structures called Miss Status Holding Registers (MSHR) or Line Fill Buffers (LFB) [20, 19] are introduced to track outstanding cache misses. There are 6-10 L1 cache MSHRs per processor core, which allow 6-10 in-flight memory requests per core, forming memory level parallelism (MLP). Specifically, the Intel's Skylake micro-architecture supports 10 L1 MSHRs and 16 L2 MSHRs.

MSHRs are usually not fully used in current software because the instruction dependency limits the issuing of enough memory accessing requests. Such a problem would be alleviated in some cases by the two techniques, hardware prefetching and speculative execution (after branch prediction). The former is able to detect and prefetch regular stridden access patterns, however it cannot hold other more complex accesses, especially for random accesses. The latter speculatively issues more but limited instructions (possibly with memory access) to execute in advance, but it wastes memory bandwidth when misprediction occurs. In all, the above two techniques still cannot fully use available MSHRs in most cases.

Software prefetching is a practical way to fully exploit MLP. It can proactively issue memory requests to move data from memory to caches before the data is needed, and the distance ahead of which a prefetching should be requested on a memory address is defined as prefetching distance. Note that the prefetching distance is controlled by programs. If the prefetching distance is too short, the requested data is not fetched into caches. But if it is too large, the requested data is evicted out of caches. Those two cases degrade the work of the software prefetching. Software prefetching with desirable prefetching distance can eliminate cache misses when the data addresses are known in advance, such as sequential accessing or looking up according to known indexing.

However, software prefetching does not work if the data addresses are not known in advance. To the operations in pointer-chasing applications, e.g., traversing skip lists, looking up hash tables and searching trees, the addresses of next nodes are not known until current nodes are processed. The time between when the addresses are known and the data are needed is much less than the desirable prefetching distance. We call the accesses with such a pattern as *immediate memory accesses*. Such accesses can only benefit from the software prefetching by taking inter-task parallelism [21]. Specifically, after a task issues memory prefetching, it will not continue because the prefetched data is not ready in time. Instead, the execution switches to other tasks, and then goes back to process the prefetched data after a while.

According to this idea, previous solutions [6, 20] share a common theme that launches G identical tasks working on different data inputs, then splits each task with N dependent memory accesses into N+1 code stages, where each stage consumes the data prefetched by the previous stage and initiates new prefetching for its next stage. How to interleave G stages from different G tasks to hide memory access latency differs in the following three approaches. Group prefetching (GP) [6] repeatedly executes N+1 stages, and each stage involves G identical stages from different tasks. Software Pipelined Prefetching (SPP) [6] runs G tasks at each stage in a pipelined fashion. Since GP and SPP couple G stages in a group or in a pipeline, they are not flexible with dealing with irregular memory access patterns. To this end, Asynchronous Memory Access Chaining (AMAC) [20] encodes N+1 stages of a task as a Finite State Machine (FSM). G running instances of the FSM execute independently but interleavingly, and each running instance goes forward to next one state circularly. AMAC is the state-of-the-art technique to exploit the software prefetching for immediate memory accesses, especially in irregular pointer-chasing applications.



(a) Probe on a chained hash table

(b) FSM of a probe

(c) FSM of a fully vectorized probe

Figure 4: Hash join probe and its FSMs (H: Hash, M: Match)

3. COMBINING PREFETCHING AND SIMD

Previous studies take either SIMD or prefetching to optimize a single operator, instead of completely combining both of them. Combining SIMD and prefetching in an operator can achieve better performance, because it can reduce branch misprediction, computation overhead, as well as cache misses. To achieve this goal, it is a straightforward approach to apply SIMD to existing prefetching-optimized techniques. SIMD can be exploited directly or fully, and the prefetching-optimized techniques include GP, SPP and AMAC. There are six cases in total that need to be discussed. For the sake of simplicity, AMAC is chosen as an example to be vectorized directly or fully in detail, and the vectorization of GP and SPP is briefly discussed, because AMAC is able to handle irregular memory accesses.

The following analysis is conducted on the hash join probe, a typical pointer-chasing application, and its results still stand on other similar applications. The hash join probe is illustrated in Figure 4(a), which is similar to that in the paper proposing AMAC [20]. The probe works on a chained hash table, where each hash bucket may contain many nodes due to hash collisions, and each node is composed of one tuple and a pointer of its next node. In the hash join probe, a tuple is sequentially fetched from a relation table, then probes the hash table in two steps: (1) computing the hash value of the join key in the tuple to find the address of a corresponding bucket; (2) iteratively matching the join keys from the tuple and each node of the bucket. Such probing process is mapped in an FSM in Figure 4(b). We can see that n times of matching causes random access of n nodes of a bucket, which is the performance bottleneck of the probe.

3.1 Direct Vectorization in Prefetching

AMAC maps the logic of a procedure into an FSM, where its states are split by immediate memory accesses, then interleaves multiple running instances of the FSM. Following such a main idea, SIMD can be directly applied to vectorizing each state of an FSM, forming directly vectorized AMAC (DVA). Note that SIMD works on a batch of tuples rather than only one tuple. The batch size equals to the number of vector lanes (W) of a vector. Particularly, a vectorized state is done only when a batch of tuples in the state are finished processing, then it can transfer to other states. In other words, a vectorized state is still active even if only one of the batch is not finished processing. It should guarantee completion of all W tuples, then goes forward to its next state.

The directly vectorized AMAC probe (DVAP) is shown in Listing 1, where each vectorized variable is prefixed with a $v\_$ prefix. It processes a batch of tuples in lockstep at each state. The matching state first loads W tuples and computes their hash values, then prefetches needed hash bucket nodes and points to its next state. The matching state iteratively compares join keys and prefetches its next bucket nodes. It should terminate all matches of current W tuples before returning to the hashing state. In fact, the matching of

W tuples may not finish at the same time due to the differing sized hash buckets. As a result, each vector in the matching states may have inactive lanes due to the early finished matches, wasting the available vector lanes.

Such inactive vector lanes are further illustrated in Figure 5(a), supposing the group size is two and each vector has four lanes to accommodate four tuples. At first, four tuples *a-d* are loaded in the left vector (i.e., H_0) to compute hash values, and H_1 computes the hash values of tuples *e-h* in the right vector. Then the two vectors shift to the matching state, i.e., M_2-M_6. Note that tuples *a-h* have 1, 2, 1, 3, 1, 1, 2 and 1 candidate matching nodes in corresponding hash buckets. After the matching state, the two vectors load new tuples *i-p* to proceed. Even though such interleaving in the two vectors can overlap computation and memory access, there are still many bubbles, i.e., the white vector lanes, denoting inactive processing. For example, in the operation M_6, the left vector has three inactive lanes due to early termination of matches from T_a, T_b and T_c.

```
1  struct fsm_t{v_key, v_payload, v_ptr /* ptr of bucket nodes*/,
       state};
2  void dva_probe(tuple_t* tuple, hashtable_t* ht, table_t* out){
3    fsm_t fsm[G]; all_done = 0;
4    while(all_done < G) {
5      k = (k == G) ? 0 : k;
6      switch(fsm[k].state) {
7        case H:{ // hash the input key, prefetch buckets
8          if(i < tuple_num) {
9            fsm[k].v_key = load(tuple[i].key);
10           v_hashed = HASH(fsm[k].v_key);
11           fsm[k].v_ptr = ht→buckets + v_hashed;
12           v_prefetch(fsm[k].v_ptr);
13           fsm[k].state = M;
14           i += W; //suppose tuple_num % W = 0
15         } else {
16           fsm[k].state = D; // the fsm is done
17           ++all_done;
18         }
19       }break;
20       case M:{ // match join keys, prefetch next bucket nodes
21         m_match = fsm[k].v_ptr→v_key == fsm[k].v_key;
22         out[num] = store(fsm[k].v_ptr→v_payload, m_match);
23         num += |m_match|;
24         m_valid = fsm[k].v_ptr→next == v_null;
25         if(m_valid) {
26           v_prefetch(fsm[k].v_ptr→next, m_valid);
27           fsm[k].v_ptr = (fsm[k].v_ptr→next, m_valid);
28         } else {
29           fsm[k].state = H; // initiate new probe in H
30         }
31       }break;
32     }  ++k;
33 }}
```

Listing 1: Directly vectorized AMAC probe

Such inactive lanes result from control flow divergence, which generally happens when a vector of data elements encounter conditions in branches or loops but cause different results. For example, the divergence in the matching state of DVAP lays at line 24 in Listing 1, which judges whether a vector of next pointers are NULL or not. Such divergence underutilizes the data parallelism provided by SIMD. In the worst case, only one lane is active in a vector, degrading the vectorized processing to the scalar execution. Additionally, such divergence prevents DVA from making full use of MLP, because the inactive vector lanes do not issue useful memory accesses.

Such control flow divergence from irregular memory accesses seriously worsens GP and SPP. Specifically, each iteration of GP and SPP executes the same or different vectorized states of G running instances of an FSM. If any running instance of the FSM terminates earlier, then the corresponding entire vector becomes idle and cannot load new tuples to continue, so directly vectorized GP and SPP waste more vector lanes compared with DVA. For example, when applying GP to the case in Figure 5(a), H_7 has to be deferred to the next iteration, and an idle vector conducts the matching state, because another running instance does not finish at that iteration.

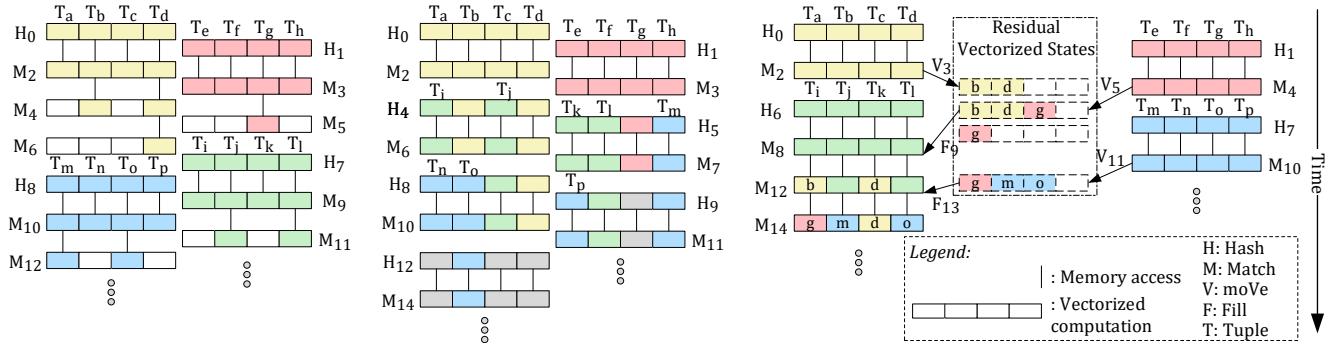
3.2 Full Vectorization in Prefetching

Fortunately, such divergence can be avoided in the full vectorization, which is introduced to vectorize most operators in databases using SIMD, including the hash join probe [28, 29]. To achieve the full vectorization for the probe, the FSM of a probe $HMM...M$ in Figure 4(b) is changed to $H\vec{M}...H\vec{M}$ in Figure 4(c) so that all states in a probe are repeated regularly, regardless of the variable size of hash buckets. Consequently, this fully vectorized probe can avoid the divergence from the loop of matching. After each matching state, the inactive vector lanes are replaced with new probing keys in the next hashing state, so there are no any idle vector lanes during execution.

Such full vectorization can be applied to AMAC to avoid the divergence problem, forming fully vectorized AMAC (FVA). States in FVA are repeated regularly so that inactive vector lanes are refilled in time. For example, as depicted in Listing 2, the fully vectorized AMAC probe (FVAP) repeats the two states, i.e., hashing and matching, in a group of running instances of the hash join probe. In contrast to DVAP in Listing 1, the matching state in FVAP is just conducted once then shifted to the hashing state, instead of repeating *n* times as the size of hash buckets in DVAP. The hashing state also selectively loads probing keys to fill inactive vector lanes in FVAP, rather than loading *W* new probing keys in DVAP.

```
1  struct fsm_t{v_key, v_payload, v_ptr, state, m_valid};
2  void fva_probe(tuple_t* tuple, hashtable_t* ht, table_t* out){
3    fsm_t fsm[G]; all_done = 0;
4    while(all_done < G) {
5      k = (k == G) ? 0 : k;
6      switch(fsm[k].state) {
7        case H:{ // hash the input key, prefetch buckets
8          if(i < tuple_num) {
9            fsm[k].v_key = load(tuple[i].key, !fsm[k].m_valid);
10           v_hashed = HASH(fsm[k].v_key);
11           v_ptr = ht→buckets + v_hashed;
12           fsm[k].v_ptr = (v_ptr, !fsm[k].m_valid);
13           v_prefetch(fsm[k].v_ptr);
14           fsm[k].state = M;
15           i += |!fsm[k].m_valid|;
16           fsm[k].m_valid = !fsm[k].m_valid || fsm[k].m_valid;
17         } else {
18           fsm[k].state = D; // the fsm is done
19           ++all_done;
20         }
21       }break;
22       case M:{ // match join keys, prefetch next bucket nodes
23         m_match = fsm[k].v_ptr→v_key == fsm[k].v_key;
24         out[num] = store(fsm[k].v_ptr→v_payload, m_match);
25         num += |m_match|;
26         fsm[k].m_valid = fsm[k].v_ptr→v_next == v_null;
27         fsm[k].v_ptr = (fsm[k].v_ptr→v_next, fsm[k].m_valid);
28         fsm[k].state = H; // add new probe in H
29       }break;
30     }  ++k;
31 }}
```

Listing 2: Fully vectorized AMAC probe



(a) Directly vectorized AMAC probe (b) Fully vectorized AMAC probe (c) Interleaved multi-vectorizing probe

Figure 5: Execution patterns of three interleaved approaches on the hash join probe

The execution pattern of FVAP is shown in Figure 5(b), where the workloads are the same as those in Figure 5(a). At first, tuples *a-h* repeat the hashing state and the matching state, but after M_2, T_a and T_c finish their matches, their corresponding vector lanes are then replaced with T_i and T_j. These two new tuples work together with original two tuples T_b and T_d. In this way, all new inactive vector lanes are filled with new probing keys of subsequent tuples. In consequence, there are no bubbles in Figure 5(b) compared with Figure 5(a), even though suffering from the control flow divergence.

Figure 5(b) shows the full vectorization can eliminate idle vector lanes, but it induces redundant hash computation for each match of tuples except the first match. For example, T_d only has three matches but brings in three times of hash, the latter two of which are unnecessary. This is essentially because FVAP changes the original FSM of the probe in Figure 4(b) (i.e., $HMM...M$) to the regular FSM of the probe in Figure 4(c) (i.e., $H\vec{M}...H\vec{M}$). As a result, such redundant hashing computation may offset the benefits from the full vectorization in contrast to the direct vectorized execution in Figure 5(a). In fact, the redundant computation is another way to waste vector lanes.

Additionally, the full vectorization is hard to be exploited in complex FSMs in following two cases. (1) An FSM has more divergent states, e.g., the FSM generated from a pipeline. Suppose an FSM has six serial states *ABCDEF*, where *C* and *E* may induce divergence. When its execution in *E* meets divergence, the execution returns to *A* to fill inactive vector lanes; but when it enters *C*, divergence may happen again, then it has to return to A again to fill new inactive vector lanes. At this point, the valid data in active vector lanes may come from state *E* or state *C*, which should be separately protected from being overrided or modified in later processing. Besides, such protected vector lanes introduce redundant operations to keep pace with the following processing. In summary, such recursive filling not only requires complex control but also seriously wastes vector lanes. (2) An FSM comes from an application with branches, like the projection operator in databases containing *case...when...* or *if...else...* statements. When a vector of data elements meet more branches, if the execution attempts to use an entire vector in any branch, then it returns to fill unqualified vector lanes that enter other branches. Note that these unqualified lanes are not invalid so that they cannot be overridden, thereby leaving no way to completely fill the vector.

We could easily apply the full vectorization to GP and SPP for the regular probe. Similar to FVAP, this eliminates idle vector lanes but induces redundant hash computation. It is more difficult to handle complex FSMs as GP and SPP that require execution in a group or a pipeline.

4. INTERLEAVED MULTI-VECTORIZING

The previous section illustrates how to combine the widely used SIMD vectorization principle with the prefetching-optimized techniques, but it neither nicely solves the divergence problem nor is applied to complex but general FSMs. To this end, we propose a novel technique called interleaved multi-vectorizing (IMV) to fully use SIMD vectorization and prefetching. IMV splits a vectorized program into multiple states wherever the program meets the control flow divergence or immediate memory accesses, forming a vectorized FSM. Then it interleaves the execution of vectorized states from those running instances of the FSM. Besides, IMV uses a residual vectorized state to solve the divergence within a state.

4.1 Interleaved Execution

To hide memory access latency, modern processors provide simultaneous multi-threading (i.e., SMT, also called hyper threading (HT)) to run several threads per physical core. Typically, a modern CPU core supports two logical threads. When one logical thread is stalled on memory accesses, another can use instruction units. However, the two interleaving logical threads of SMT are too few to hide memory access latency, especially in memory-intensive applications. Such deficiency still stands in vectorization, even though each thread issues more memory accesses. Therefore, it is also necessary to interleave more instruction streams in vectorization, letting more instruction streams hide cache misses among them.

The interleaved execution requires an efficient way to suspend and resume each instruction stream. It can be achieved in user space by the following two ways. (1) Coroutine [19, 16, 31]. It is a "resumable function" that can suspend its execution and return a handle to its caller before it completes, then the caller uses the handle to resume the function. So several coroutines can be interleaved by scheduling their suspension and resumption. However, the coroutine is not available in the current C++ standard. It is under review to become a part of the C++20 standard. Some experimental features are explored in [16, 31], but this method does not outperform the implementation based on GP or AMAC in pointer-chasing applications. To this end, we adopt the second way below and defer the discussion about combining coroutines and SIMD to Section 6. (2) Interleaved FSMs [20]. Although current C++ does not support the suspension and resumption mechanisms, they can be manually implemented in programs. Taking AMAC for example, a program is split into many states wherever it issues immediate memory accesses, forming an FSM. After a state issues memory access requests through the software prefetching, it is suspended by storing its contexts to a circular array, and another state of other running instances of the FSM is resumed by restoring its contexts from the circular array. In this way, the memory access latency

from an instruction stream can overlap with the computation from other instruction steams. In particular, the context switch among states induces less time than a memory stall. Moreover, this way in AMAC performs well in pointer-chasing applications, so we adopt this method to interleave multiple vectorized programs.

4.2 Splitting States for a Vectorized Program

According to the principle of interleaved FSMs, a vectorized program is split into a series of states, forming a vectorized FSM. During the interleaved execution, after a vectorized state ends with requesting accessing memory data through prefetching, it does not continue to its next states but switches to the state of other running instances of the vectorized FSM. In such a way, a group of interleaved running instances of the FSM are suspended and resumed in turn, and each running instance has its own running contexts. The group size should be large enough so that the interleaved computation can overlap with memory accesses. We will test the optimal group size through experiments.

However, the vectorized states suffer from the control flow divergence as analyzed earlier in Section 3. Even worse, going either forward or back with the non-full vectors cannot make full use of DLP and MLP in the direct or full vectorization. The essence of the problem is that the non-full vectors from current execution have an impact on the subsequent processing, so it is necessary to prevent the non-full vectors of current execution from involving in the next processing. To achieve this goal, we further split a vectorized state into several smaller states wherever meeting divergence; the divergence of each smaller state is then solved within the state by the residual state vectors below. Finally, any divergent state would not pollute its next states. Such splitting from divergence guarantees each vectorized state fully utilizes available vector lanes.

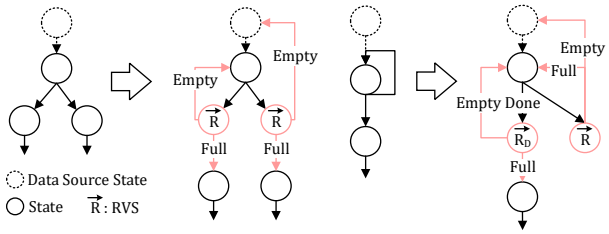
4.3 Residual Vectorized States

To solve the divergence within a vectorized state, we propose attaching a residual vectorized state (RVS) with each divergent vectorized state (DVS), which stores the contexts of the DVS with non-full vectors. Before a DVS continues to its next states, it integrates with its RVS. If the active vector lanes in the DVS and its RVS are not less than the size of an SIMD vector (i.e., W), the RVS will fill the DVS. Consequently, a new fully vectorized state can go forward to the next state while the remaining active lanes reside in the RVS. Otherwise, the active vector lanes in current DVS are moved to its RVS, leaving an empty state that will return to a previous state to restart new execution, i.e., its nearest data source state. The integration between a DVS and its RVS is shown in Listing 3. After the integration in such two cases, the DVS becomes full or empty, so current non-full vectors do not pollute subsequent processing. To this end, each vectorized state fully utilizes DLP.

```
1  if(DVS_active_lane_cnt + RVS_active_lane_cnt < vector_size) {
2      DVS = compress(DVS); // left packing valid lanes in DVS
3      RVS = expand(RVS, DVS); // filling RVS
4      // return to a previous state to restart new execution
5  } else {
6      DVS = expand(DVS, RVS); // filling DVS
7      RVS = compress(RVS); // left packing valid lanes in RVS
8      // go to its original next state
9  }
```

Listing 3: The integration between a DVS and its RVS

After the RVS being introduced, any DVS has to integrate with its RVS before continuing its subsequent processing, thus changing the original FSM. Such change is analyzed for *if* and *loop* statements, respectively. For an *if* statement in vectorization, more than



(a) A branch statement (b) A loop statement

Figure 6: Updated FSMs

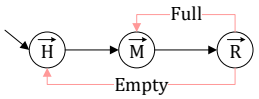


Figure 7: The FSM of the IMV probe

one of the branches may be processed after evaluating conditions. Before entering any branch, a residual vectorized state is added to update the divergent vectorized state from the conditions. This change is shown in Figure 6(a). After the integration in the RVS, if vectors in the DVS become full, the execution continues to its next state. Otherwise, the active vector lanes in the DVS are moved to the RVS, leaving empty vectors. After these two cases, the execution returns to sequentially process other branches in a similar way. Note that after the last branch is completed, the execution goes back to its nearest data source state, restarting new execution. However, such sequential processing in branches becomes sophisticated if the branch body contains complex statements, like *if* or *loop* statements. In such a case, the full DVSs after integration are stored in a task queue, instead of being processed immediately, then each task in the queue will be executed by coming idle running instances. With regards to a general *loop* statement illustrated in Figure 6(b), after an iteration of the loop, a current state becomes divergent, then this divergent state can be processed according to the way in the *if* statement.

RVSs cost little overhead in space and time during the interleaved execution, because each RVS is shared among a group of running instances of the DVS. The number of RVS instances equals the number of DVSs of an FSM. They cost less vectors thus reside in memory caches. In particular, all instances of RVSs are owned by the current running instance of an FSM, and their ownership changes with the switching of running instances of an FSM. In other words, each RVS is sequentially read and written by a running instance of an FSM, so there are no conflicts among read and write in an RVS. Additionally, the integration between RVSs and DVSs is processed among vectors, thus its overhead is far less than accessing memory.

4.4 Example Analysis

The key idea of IMV is to interleave vectorized states from multiple running instances of an FSM. Each instance independently processes a batch of data. Thus IMV can be used in once-a-batch execution model in databases. In particular, IMV is able to speed up pointer-chasing applications, such as traversing skip lists, searching trees and looking up hash tables. Besides, it can directly accelerate a series of operators in a query pipeline, forming a fully SIMD vectorized execution model. Here, we take the hash join probe and a pipeline as examples to illustrate how IMV works.

The FSM of the IMV probe is illustrated in Figure 7. The matching state may become divergent, so an RVS is added. After the integration in RVS, the vectors may be fully filled with valid data to

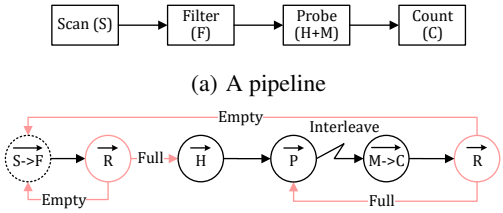
do matching again, or become empty after moving valid data to the RVS, and then return to the hashing state. Such integration is the main difference between the IMV probe and the DVA probe. So we just add an RVS to Listing 1, then the DVA probe is changed to the IMV probe, which is shown as Listing 4. The integration is added at line 26 when divergence may happen at the previous line. After integration, if the FSM instance continues to do matching (i.e., fsm[k] is full), then it will prefetch data; otherwise, it will directly shift to the hashing state. In particular, if there are many empty buckets in the hash table, the state is not full before entering the matching state. Such divergence requires adding a new state to check valid buckets after the hashing state, but this is omitted here for simplicity.

```
1  struct fsm_t{v_key, v_payload, v_ptr, state, m_valid};
2  void imv_probe(tuple_t* tuple, hashtable_t* ht, table_t* out){
3    fsm_t fsm[G], RVS;  all_done = 0;
4    while(all_done < G) {
5      k = (k == G) ? 0 : k;
6      switch(fsm[k].state) {
7        case H:{ // hash the input key, prefetch buckets
8          if(i < tuple_num) {
9            fsm[k].v_key = load(tuple[i].key);
10           v_hashed = HASH(fsm[k].v_key);
11           fsm[k].v_ptr = ht→buckets + v_hashed;
12           v_prefetch(fsm[k].v_ptr);
13           fsm[k].state = M;
14           fsm[k].m_valid = 0xFF;
15           i += W; //suppose tuple_num % W = 0
16         } else {
17           fsm[k].state = D; // the fsm is done
18           ++all_done;
19         }
20       }break;
21       case M:{ // match join keys, prefetch next bucket nodes
22         m_match = fsm[k].v_ptr→v_key == fsm[k].v_key;
23         out[num] = store(fsm[k].v_ptr→v_payload, m_match);
24         num += |m_match|;
25         fsm[k].m_valid = fsm[k].v_ptr→next == v_null;
26         integration(fsm[k], RVS);
27         if(fsm[k].state==M) {
28           v_prefetch(fsm[k].v_ptr→next, fsm[k].m_valid);
29         } else {
30           k = k−1; // directly shift to H
31         }
32       }break;
33     }  ++k;
34 }}
```

Listing 4: Interleaved multi-vectorizing probe

The execution pattern of the IMV probe is shown in Figure 5(c). The data distribution is the same as that in Figure 5(a). The main difference lays at the integration. After M_2, T_a and T_c terminate, T_b and T_d are left active in the two vector lanes. At this time, T_b and T_d are moved to the residual vectorized state of the matching state, then a fully empty vector loads subsequent tuples *i-l* to process. Similar cases happen after M_4 and M_{10}. After M_8, the remaining two active lanes of T_j and T_l plus the three active lanes in its RVS can fully fill a vector, so T_b and T_d are reloaded to the vector. Then prefetching instructions are issued, subsequently M_{12} occurs. An analogous case happens in M_{14}. Figure 5(c) demonstrates that IMV fully utilizes DLP and MLP in contrast to Figure 5(a) and Figure 5(b).

IMV provides a mechanism to benefit from prefetching in vectorized programs, instead of only in one operator. Next, we illustrate how IMV works in a query pipeline that suffers from the control flow divergence and heavy memory accesses. An example pipeline is shown in Figure 8(a). In the pipeline, tuples come from the scan



(a) A pipeline

(b) The FSM of the pipeline (P: Prefetching)

Figure 8: A pipeline example using IMV

Table 1: Details of hardware configurations

	SKX	KNL
Cores	8	64
Threads	2 threads/core	4 threads/core
Frequency	2.10GHz	1.05GHz
L1d/L1i cache	32KB/32KB	32KB/32KB
L2 cache size	1MB	1MB
L3 cache size	11MB	NA
Memory size	150GB	96GB
L1 TLB entries for 2MB huge pages	32	128
SIMD	512 bits	512 bits

operator, but only some of them can pass the predicates in the filter operator, the passed tuples then compute hash values and match a hash table in the probe operator, lastly counting qualified tuples.

In IMV, the pipeline is transformed to the FSM in Figure 8(b). As a vector of tuples suffer from the control flow divergence after the filter, an RVS is appended. After the hashing, however, the matching loop not only accesses memory data but also encounters divergence in each iteration, so a prefetching state and an RVS are added before and after the matching state. Note that after the prefetching state, one interleaved execution will be issued, then the current running instance of the FSM is suspended and another running instance of the FSM will be resumed. The two RVSs instead transfer to other states within the current running instance.

5. EXPERIMENTS

In this section, we first compare IMV with other competitors on four individual operators. Then we test all approaches on Xeon Phi, a many-core co-processor. Finally, we apply IMV to execute a query and compare its performance against those of three popular execution models.

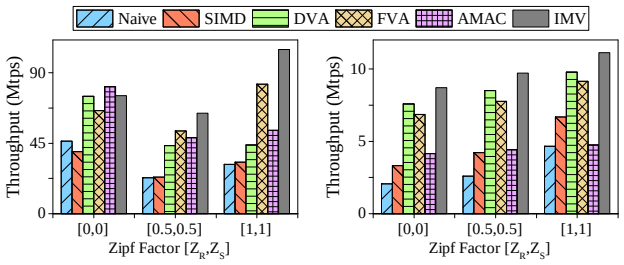
5.1 Experimental Setup

We conduct experiments on two hardware platforms: a server equipped with two Intel Xeon Silver 4110 CPUs based on Skylake micro-architecture (SKX), and a server with an Intel Knights Landing processor 7210 (KNL). The hardware specifications of SKX and KNL are listed in Table 1. Our code is compiled by *gcc 6.4.0* with *-O3* optimization enabled. The affinity of threads is set to avoid scheduling overhead, and prefetching data is implemented through the *_mm_prefetch()* instruction with *_MM_HINT_T0* hint. The source code used for all experiments is available online [1].

Workloads. The experiments on individual operators are conducted on synthetic workloads, which are generated according to the method shown in [4]. All workloads involve two relations (R and S). Each tuple of tables contains an 8-byte integer key and an 8-byte integer payload. They are distributed uniformly or not by controlling Zipf factors. The Zipf factor of relations R and S is denoted by [Z_R, Z_S], and Zipf $\in [0, 1]$. Especially, [0,0] denotes uniform data distribution. Note that keys of S are in the range of R to guarantee equivalent keys in the two relations.

Table 2: The speedup of IMV over others

Configurations	Naive	SIMD	DAV	FVA	AMAC
HJP,[0,0],1 thread	1.62	1.91	1.01	1.15	0.93
HJP,[0.5,0.5],1 thread	2.79	2.76	1.48	1.22	1.33
HJP,[1,1],1 thread	3.34	**3.17**	**2.39**	**1.27**	1.97
BTS,[0,0],1 thread	**4.23**	2.62	1.15	1.27	2.10
BTS,[0.5,0.5],1 thread	3.76	2.30	1.14	1.25	2.21
BTS,[1,1],1 thread	2.39	1.66	1.14	1.22	**2.34**
HJP,[0,0],32 threads	1.62	1.37	1.01	1.14	1.00
HJP,[0.5,0.5],32 threads	2.09	1.71	1.31	1.11	1.15
HJP,[1,1],32 threads	1.66	1.79	**1.89**	**1.49**	1.54
BTS,[0,0],32 threads	**2.76**	**1.86**	1.15	1.26	1.85
BTS,[0.5,0.5],32 threads	2.31	1.52	1.13	1.23	1.83
BTS,[1,1],32 threads	1.52	1.22	1.14	1.26	**1.97**



(a) Hash join probe (b) Binary tree search

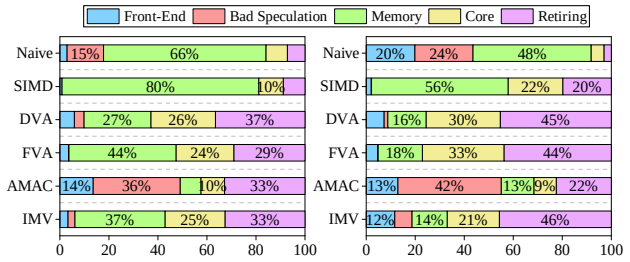
Figure 9: The impact of data distribution

For all techniques and algorithms, we carefully tune their code and deployment to use the best performing parameters in all experiments. All experiments on individual operators use the modulo hashing, except for the experiment in Section 5.5 that takes the *MurmurHash3* in [17]. By default, to avoid TLB misses, experiments take 2 MB huge pages, and choose 1 M (1 M = 2^{20}) tuples in relation R and 50 M tuples in relation S. We set optimal group sizes for all interleaving executions, 5 for IMV, FVA and DVA, 20 for AMAC. The performance is measured by the throughput of approaches, i.e., million tuples per second.

5.2 Hash Join Probe and Binary Tree Search

The hash join probe (HJP) and the binary tree search (BTS) share the similarity of finding matches in a data structure. HJP adopts the chained hash tables optimized in the previous work [4]. Each hash table bucket may contain more than one node due to hash collisions. Each node is composed of a 1-byte latch for synchronization, a 16-byte tuple and an 8-byte pointer to its next node. In particular, the hash join probe writes the payloads of matched tuples to a buffer, as that in the once-a-batch execution model. The binary tree takes a typical implementation, which is built from relation S. Each node contains a 16-byte tuple and two 8-byte pointers to its right and left children. The key of a node is bigger than that in its left child but is less than the key in its right child. Besides, there are no duplicated keys in the tree. The tree is searched by tuples from relation R to find equal keys. If one tuple matches a node in the tree, then their payloads generate a new tuple to be stored in a buffer, resembling an index join.

We test the performance of HJP and BTS in the six approaches using one thread or all threads, and experiments are conducted under various data distributions on SKX. The single-threaded results are presented in Figure 9, and the speedup of IMV over others is listed in Table 2. IMV outperforms others in most cases, up to 4.23X, 3.17X, 2.39X, 1.27X and 2.34X faster than Naive (i.e., the pure scalar implementation), SIMD (i.e., the pure SIMD implementation), DVA, FVA and AMAC, respectively. We further analyze



(a) SKX, HJP, [1,1] (b) SKX, BTS, [1,1]

Figure 10: Execution time breakdown based on microarchitectural analysis

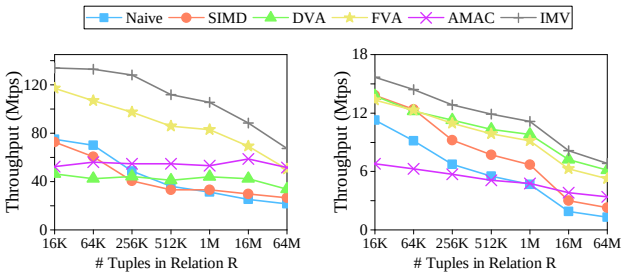
the advantages of IMV through micro-architectural metrics. The methodology is specified by TMAM [2]. The execution time breakdown reported by Intel Vtune is depicted in Figure 10. 'Front-End' represents a fraction of slots during which CPU was stalled due to front-end latency issues, such as instruction-cache misses, ITLB misses or fetch stalls after a branch misprediction. 'Bad Speculation' represents a fraction of pipeline slots wasted due to incorrect speculations. 'Retiring' means a fraction of pipeline slots utilized by useful work, i.e., excluding the fraction from 'Bad Speculation'.

Figure 10 shows why IMV is significantly faster than others. IMV not only reduces memory access costs but also eliminates bad speculations (i.e., branch misses). In Naive, both HJP and BTS are dominated by memory accesses (66% and 48%, respectively). Although optimizing memory accesses in AMAC can speed up performance, this method is still limited by serious bad speculations (36% and 42% in HJP and BTS, respectively). The results of Naive and AMAC show, only eliminating branches in SIMD almost does not work because of numerous cache misses (80% and 56% in HJP and BTS, respectively), so we conclude that only reducing memory accesses or eliminating branches is not enough to significantly accelerate HJP and BTS, as well as similar applications. Fortunately, those two goals can be achieved by combining vectorization and prefetching in DVA, FVA and IMV. IMV is superior to DVA and FVA because IMV provides a better way to eliminate divergence in vectorization as analyzed in Section 4.

In addition, we evaluate the performance of the six approaches in HJP and BTS using all threads. We adopt the morsel-driven [24] parallelism to dispatch data to each thread, and since HJP is sensitive to the NUMA architecture (see Figure 14(a)), we copy related data in all sockets to avoid remote accesses. The speedup of IMV over others is also listed in Table 2. IMV achieves up to 2.76X, 1.86X, 1.89X, 1.49X and 1.97X better performance than Naive, SIMD, DVA, FVA and AMAC approaches, respectively. In all configurations, the speedup using all threads is almost slower than that using a thread, because more threads compete for shared resources, including memory bandwidth, caches, VPUs and TLB entries. The performance of those six techniques differs in various configurations, which will be further analyzed in next subsections.

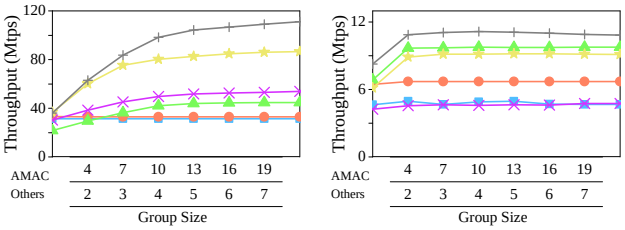
5.2.1 Parameters of Workloads and Techniques

Data Distributions. We evaluate the impact of data distributions by changing the Zipfian factor as in [20]. The results are shown in Figure 9. For HJP, AMAC achieves the best performance in [0,0], because it rarely encounters branch misses on uniform datasets, while IMV, FVA and DVA pay additional efforts to solve divergence. However, AMAC becomes much worse on skew datasets, especially in [1,1]. DVA is inferior to IMV and FVA in skew datasets because it suffers from heavy divergence. BTS differs in that it is not influenced by the data distribution, because the built tree is bushy and without long chains. Due to less divergence, DVA is a bit faster than FVA, but it is still slower than IMV. The three



(a) SKX, HJP, [1,1] (b) SKX, BTS, [1,1]

Figure 11: The impact of data size



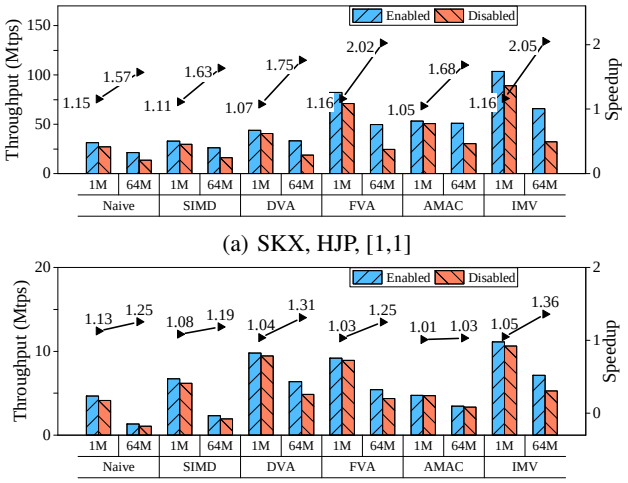
(a) SKX, HJP, [1,1] (b) SKX, BTS, [1,1]

Figure 12: Group size

are superior to AMAC because AMAC suffers from heavy branch misses. Thanks to this, AMAC is even slower than SIMD in [1,1]. Although SIMD can reduce heavy branch misses, it cannot alleviate cache misses, so it is inferior to FVA, DVA and IMV. In this experiment, we observe taking either SIMD or the software prefetching is not enough to speed up performance, while combining both of them can significantly boost performance in those applications.

Data Size. We study the impact of data size on various approaches. We vary the tuples of relation R from 16 K (1 K = 2^{10}) to 64 M. The tuple number multiplying by the size of hash bucket nodes or tree nodes (32 bytes in both data structures) gets the total data size. The results of [1,1] data distribution are demonstrated in Figure 11, and other cases get similar plots. In Figure 11(a), with increasing tuples in relation R, the throughput of all approaches except AMAC in HJP decreases. This is because AMAC is seriously dominated by branch misses verified in Figure 10(a), while others suffer from more cache misses with increasing data. SIMD and Naive drop faster when the tuple number is larger than 256 K, because beyond that the data is out of the memory cache. Since DVA suffers from heavy divergence, it cannot benefit from interleaving and vectorization but instead sustains more overhead from them. This is the reason why DVA is the slowest one when data resides in cache. Furthermore, when tuples are larger than 1 M, all techniques suffer from more TLB misses. With regards to BTS in Figure 11(b), the performance of all techniques is degraded as the data size increases, especially for SIMD and Naive, due to more branch misses, TLB misses and computation overhead.

Group Size. The effect of prefetching depends on the prefetching distance. It is controlled by the group size of all interleaving executions, i.e., the number of running instances of an FSM. It should be large enough so that there is sufficient computation to overlap memory access latency. In fact, it is also limited by the number of MSHRs per core, beyond which more data accesses are blocked. Generally the number of memory access requests in a group are larger than that of MSHRs, because some requested data elements of a group may be hit in the cache at run time. In addition to the uncertain number of misses in a group, the computation overhead between two memory accesses in a running FSM is unequal, so it is hard to predict the group size in advance. Here we choose the optimal group size through experiments.



(a) SKX, HJP, [1,1]

(b) SKX, BTS, [1,1]

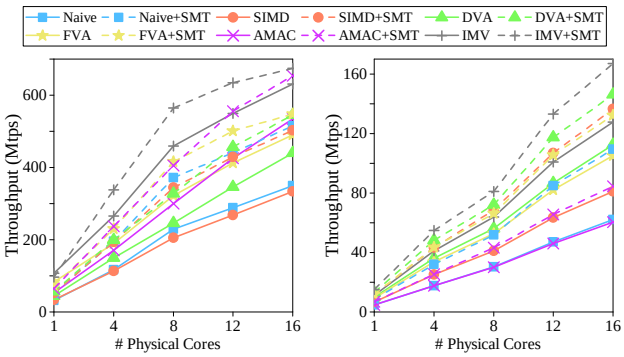
Figure 13: The impact of huge pages

Figure 12 shows the impact of group sizes in various configurations (All get similar plots, some of which are not shown here due to limited space). We observe that in all subfigures the throughput of interleaving approaches increases with the group size, but such growth stops after a certain point. Such a sweet point varies on different platforms, data distribution and applications. After the sweet point, the performance is nearly not changed. The corresponding group size of the sweet point or the larger value can be chosen as the optimal group size in the configuration. For the interleaving vectorized algorithms IMV, FVA and DVA, we can set the optimal group size as five (G = 5) in all configurations. This value implies at most 40 memory access requests are issued in a group of running instances of an FSM (A vector in the group operates 8 64-byte elements at a time). They are large enough to occupy MSHRs in most cases at run time. Larger group size increases the overhead of loading and storing intermediate vectorized states, slightly decreasing the performance. With regards to AMAC, the optimal group size can be set as 20. SIMD and Naive are not effected by varying group size.

5.2.2 *Parameters of System Architecture*

Huge Pages. Physical memory is segmented into a series of contiguous regions called pages. Each page contains a number of bytes, referred to as the page size, 4 KB by default. Each page has a virtual address, which should be mapped to a physical address when accessing data in the page. Such mapping time is reduced by caching the recent translations of virtual addresses to physical addresses in an address-translation cache, called translation lookaside buffer (TLB). The TLB is of limited size. When it does not contain the physical address of a requested virtual address, a TLB miss happens, then the translation proceeds by looking up the page table. This is time-consuming compared with a TLB hit, which means a virtual address has a corresponding physical address in TLB. The costly TLB misses can be slightly avoided by larger page sizes, because a TLB cache of the same size can keep track of larger amounts of memory of huge pages. We set 2 MB huge pages in SKX, which contains 32 TLB entries in L1 cache. This still cannot meet the demand of large memory applications. We test the influence of TLB misses by enabling or disabling huge pages, as well as changing the data size of relation R.

As illustrated in Figure 13, enabling huge pages accelerates the performance in both HJP and BTS. The speedup is larger with increasing data size. We conclude that TLB misses play an important



(a) SKX, HJP, [1,1] (b) SKX, BTS, [1,1]

Figure 14: The scalability of all approaches

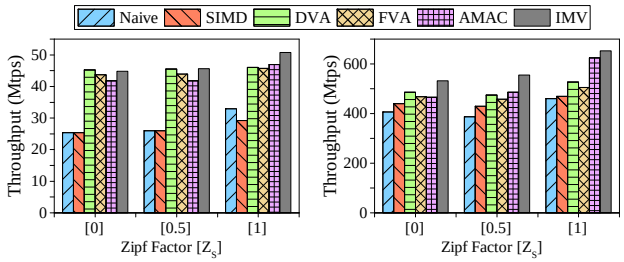
role in the two applications, especially for a larger dataset. Specifically, 1 M tuples in relation R occupy almost 32 MB memory in the hash table and the binary tree. Such a data size needs 8*1024 4-KB pages, which is far larger than the TLB entry number in L1 cache on SKX. This problem becomes worse for 64 M tuples of relation R. Even with huge pages, the TLB entries in L1 cache for 2 MB pages just meet the demand of 1 M tuples, but cannot accommodate the page identities of 64 M tuples in relation R.

Among all algorithms, enabling huge pages achieves higher speedup for IMV, especially under the larger dataset. It is up to 2.05X in HJP, because TLB misses become the main bottleneck for IMV after it reduces cache misses and branch misses with best efforts. We further find that the technique achieves higher throughput, but it suffers more effects from huge pages, because it is more likely dominated by TLB misses after eliminating other factors. On the other hand, when comparing IMV, FVA and DVA with AMAC, their speedup over AMAC becomes smaller with increasing data size. This is because the vectorized code likely refers to more pages when gathering or scattering data, prone to causing TLB misses.

Scalability. Modern processors contains many physical cores. We then evaluate how the interleaved execution benefits from more cores or threads, respectively. We present the results in Figure 14. From 1 core to 8 cores within a socket, the throughput of all approaches in HJP and BTS speeds up linearly. But trends are diverse when scaling beyond 8 cores due to the influence of the NUMA architecture. Such influence is larger on IMV in HJP, shown as Figure 14(a), especially when enabling SMT. Because in this case IMV is seriously limited by heavy remote accesses. This is the reason why the speedup of IMV over others drops when using all cores or threads. However, the performance differs in BTS shown as Figure 14(b). Beyond 8 cores, the throughput of all executions still increases linearly, because there are no large remote accesses in BTS. In the two applications, for a fixed number of cores, enabling SMT does not double the throughput in all executions. This is because the logical threads within a core compete for limited resources in the core, including bandwidth, TLB entries and registers.

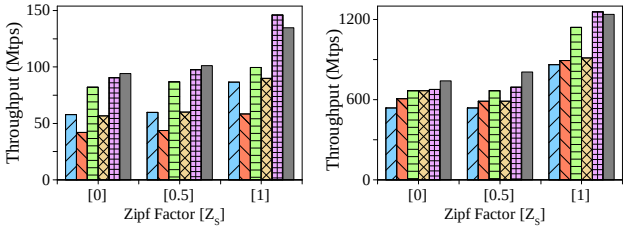
5.3 Hash Join Build and Hash Aggregation

In this subsection we measure the performance of IMV under workloads with numerous writes. The hash build is one of the two phases in the hash join. It inserts new nodes into hash buckets in the following three steps. First, it computes the hash value of a key and finds a corresponding hash bucket. Then, it applies for a piece of space and writes the key and payload to the space, forming a new node. Finally, it inserts the new node at the head of the bucket. During these steps, randomly accessing each bucket head induces numerous cache misses. The (grouped) aggregation is more complex. It probes the corresponding bucket according to



(a) 1 thread (b) 32 threads

Figure 15: The performance of the hash build



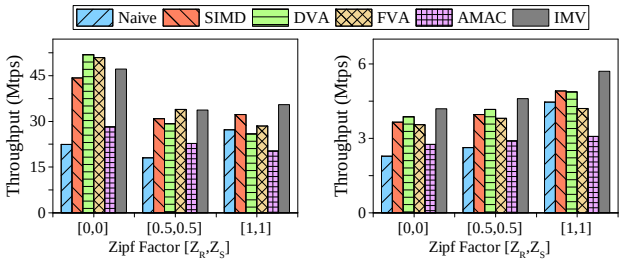
(a) 1 thread (b) 32 threads

Figure 16: The performance of the hash aggregation

the hash value. If it finds a matched node in the bucket, then it updates the aggregators in the node. Otherwise, it inserts a new node at the tail of the bucket. During this period, probing hash buckets also causes numerous cache misses. These two operators also induce a lot of writes and memory allocation while working on the same hash table in the probe. Besides, they run independently in the multi-threaded execution.

Figure 15 depicts the performance of the hash build. IMV almost outperforms Naive by 1.77X and SIMD by 1.76X in the three kinds of data distribution when using one thread, because IMV can reduce the most of cache misses in the build. Such an effect can also be achieved by DVA, FVA and AMAC, so their performance is comparable to IMV. However, under 32 threads, the benefits of the interleaved execution, i.e., IMV, FVA, DVA and AMAC, decrease. IMV just performs 1.31X, 1.44X and 1.42X faster than Naive in Zipf data distributions with 0, 0.5 and 1, respectively. This is because multiple threads of the build seriously compete for memory bandwidth and TLB entries. Besides, applying for memory results in numerous system calls. These issues are observed through Intel Vtune, but unfortunately cannot be alleviated by the software prefetching in the interleaved execution. If they can be reduced in multi-threaded execution, the build will benefit more from IMV, as in the one thread case. Since the build inserts new nodes at the head of buckets, it is also not sensitive to the data distribution in both 1 thread and 32 threads cases.

Figure 16 demonstrates the performance of the hash aggregation. IMV achieves almost 1.7X and 2.3X speedup over Naive and SIMD under the single-threaded execution in Figure 16(a). SIMD is the slowest because it repeats all steps of the aggregation according to the full vectorization, causing much redundant computation. Similar to the build, the advantages of IMV decrease in multi-threaded execution, because the aggregation is also bounded by the limited memory bandwidth and TLB entries in multiple threads. In addition, even under skewed workloads, duplicated keys are merged in the bucket, reducing the length of each bucket for probing, so the aggregation cannot benefit from IMV to a larger extent. FVA is slower than the other three interleaving approaches, because it also repeats all steps of aggregation each time, causing a large amount of redundancy overhead as in SIMD. Particularly in the '[1]' case, AMAC outperforms IMV, because such high skewed data causes heavy conflicts when inserting new nodes or updating aggregators.



(a) Hash join probe (b) Binary tree search

Figure 17: The performance on KNL

It requires a conflict-free way to implement aggregation in order to improve IMV, but this is out the scope of this paper.

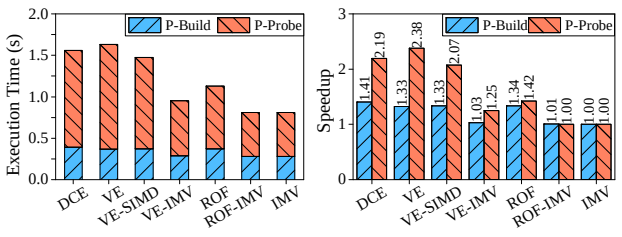
5.4 Knights Landing (Xeon Phi)

The microarchitecture of Intel's Xeon Phi is called Knights Landing, and it is different from that of Skylake. Its chip is equipped with 64 to 72 cores. Each core has two 512-bit vector processing units (VPUs). In addition, each core supports four logical threads, therefore Phi has a higher ability to hide memory access latency when enabling SMT. This differs from the interleaved execution because it is provided by hardware, while the interleaved approaches are implemented in software. We wonder whether the interleaved approaches can still work well on such a platform.

Experiments are conducted on one core enabling four logical threads to avoid the scalability issue. The group size in AMAC is reduced to 10, and 2 for DVA, FVA and IMV. Results are presented in Figure 17 (Similar results on build and aggregation , omitted due to limited space). The scalar implemented Naive and AMAC are significantly slower than others, because other four methods benefit from SIMD. Also, since more threads on a core issue more memory access requests, AMAC is a bit faster than Naive. However, AMAC is slower than Naive in [1,1] due to its interleaving and prefetching overhead. Such overhead also sometimes slows down the performance of DVA and FVA, so they are slower than SIMD in some cases, but this overhead can be offset in IMV with a better way to fully use vectors, thus IMV outperforms others, up to 2.1X faster than Naive and 1.2X faster than SIMD. The results highlight the importance of the interleaved execution on CPUs with fewer logical threads per core.

5.5 Comparison with Other Execution Models

Finally, we apply IMV to the execution of a whole query and compare against other three state-of-the-art execution models: data-centric compilation execution (DCE) [17], vecotrized execution (VE) [17] and relaxed operator fusion (ROF) [26]. We design a query based on TPC-H benchmark, which is shown below. Such a join generally occurs in TPC-H and dominates the query performance. The query's physical plan involves two pipelines, $scan \rightarrow filter \rightarrow build$ (called P-Build) and $scan \rightarrow filter \rightarrow probe \rightarrow count$ (called P-Probe). This plan is implemented on the base of [17] using different execution models. Specifically in IMV, the FSM of the probe pipeline is depicted in Figure 8, and the build pipeline gets a similar FSM. Our IMV can also be adopted in VE and ROF to accelerate the join operator, which are labeled as VE-IMV and ROF-IMV, respectively. VE can also adopt SIMD to speed up the join, denoted as VE-SIMD. The query is executed on TPC-H data with SF 100, using all 32 threads. The morsel size in DCE, ROF, ROF-IMV and IMV is 10K, the vector size in VE, VE-SIMD and VE-IMV is 1K, and the buffer size among stages in ROF is 10K. These settings work well in each model and are in line with previous work [17, 26].



(a) Execution time (b) Speedup over IMV

Figure 18: The comparison of engines

SELECT count(∗) **FROM** orders, lineitem **WHERE** o_orderkey= l_orderkey **AND** l_quantity<50 **AND** o_orderdate < '1996−1−1';

Figure 18(a) shows the execution time of the query in different execution models. IMV performs 1.92X, 2.01X and 1.39X faster than DCE, VE and ROF, respectively. Using IMV to accelerate the join operator in VE and ROF achieves 1.72X and 1.39X speedup, respectively. DCE is slower than IMV because DCE can neither reduce cache misses in the join nor avoid branch misses in the filter. This is the main reason why the build pipeline of DCE works slower than others (see Figure 18(b)). The probe pipeline of DCE is a bit faster than VE, because VE materializes immediate results among stages, causing useless computation. Specifically, VE splits the hash probe into three stages, i.e., computing hashing, generating join candidates, and checking equality. Among stages, the results are written to a vector, and the results of non-equal join candidates waste some computation, which may offset the benefits from the out-of-order execution in VE. Although such execution can be directly accelerated by SIMD, VE is still dominated by cache misses. So using IMV-optimized join in VE can improve performance to a large extent, but it is still slower than IMV due to the numerous function calls in its pull execution.

ROF takes SIMD to optimize the filter and combines the group prefetching (GP) to implement the join. GP outperforms AMAC in such uniform dataset in spite of using the chained hash table. ROF works faster than DCE and VE because it reduces a lot of cache misses. However, its performance is worse than that of IMV. Its build pipeline and probe pipeline are 1.34X and 1.42X slower than those of IMV. This is because IMV reduces cache misses as well as branch misses, so replacing GP with IMV to accelerate join in ROF works better. Also, ROF-IMV performs almost the same as IMV. The two both use SIMD to speed up the filter and take IMV to elevate the join. The difference is that ROF-IMV breaks the filter and the build or probe of the join into two stages, which are linked by a buffer. It induces materialization overhead, especially when the filter does not get rid of tuples. However, IMV combines the filter and the build or the probe together. This way may perform worse, because it may randomly take more time to get a full SIMD vector after the filter, which cannot ideally overlap with memory accessing. The performance difference of the two methods is related to the selectivity in the filter, but the performance gap is almost within 5%, so we think applying IMV to ROF is also a good choice to speed up the whole query execution.

6. DISCUSSION

IMV in Complex Queries. Complex queries like in TPC-H are composed of pipelines, which can benefit from IMV to reduce cache misses, branch misses and computation overhead. As shown in Section 5.5, pipelines can be implemented using IMV in two execution models: (1) implementing the whole pipeline using IMV; (2) linking IMV-optimized operators in a pipeline as in ROF. The first method is more complex than the second, because the first should

ask help from the Just-In-Time (JIT) compilation to fuse all operators in a pipeline. Combining SIMD and JIT is an interesting topic, which is preliminarily investigated in [9, 11], but not forming a general approach. Besides, fusing more operators in a pipeline may disturb the interleaved execution shown in Section 5.5, so we think the second is more practical for using IMV.

For each specific operator in pipelines, its complex processing logic can be handled in IMV, because IMV is able to solve the divergence from general *if* and *loop* statements. Moreover, the main logic of individual operators is almost fixed, including the positions where cache misses and the control flow divergence may occur, so it is rather easy to decide where to insert the prefetching and integration. In fact, a specific operator becomes diverse because of varying parts, like expressions and hash functions both supporting multiple columns and different data types. To deal with these varying parts, we implement various SIMD vectorized primitives that are called at runtime. This is analogous to the scalar vectorized execution [5], but the primitives are SIMD-vector-oriented, rather than scalar-batch-oriented in [5]. However, SIMD cannot directly compute complex expressions involving special datatypes (like *varchar* and *decimal*) and operations (like *substr()*). Specially, if the unique values of a complex type are not large, they can be mapped to limited integers to speed up some operations (e.g., equality checking), as in [11, 30]. In other cases, the special operations should be implemented into SIMD-like primitives using scalar code to seamlessly integrate with SIMD operations. With regards to performance, the overhead of invoking SIMD(-like) primitives cannot be sufficiently amortized due to limited tuples within a vector, but it can be alleviated using JIT [9, 27]. Furthermore, such overhead may almost overlap with memory accesses in the interleaved execution, so IMV can still achieve obvious speedup.

IMV Automation. In this paper, we manually implement the interleaving and solve the control flow divergence using the residual vectorized states. It is ideal to automate the two procedures in order to hide them from software developers. The interleaving can be achieved by coroutines [19, 16, 31]. It provides an easy way to suspend or resume the execution of a function, increasing code readability and maintainability. However, in vectorization the coroutines still have to consider the control flow divergence. If taking the residual vectorized states, the coroutines should equip three following abilities. First, coroutines should identify where the divergence would occur and the residual vectorized states are inserted into. Second, coroutines should efficiently share the residual vectorized states among coroutines at run time to reduce the requirements of vectors. Third, coroutines should automatically schedule the execution of multiple branches instead of being controlled by programs as analyzed in Section 4.3. The three requirements are critical to automatically solve the control flow divergence on CPUs.

7. RELATED WORK

SIMD in DB. SIMD has been widely studied in databases because it can reduce branch misses [13] and computation overhead, as well as provide convenient instructions to access data. Important operations, such as scans, index scans, joins, aggregations, index operations and sorting are implemented using SIMD [28, 29, 14, 10, 35, 8]. In particular, the probe of a two-table join using the cuckoo hash [33] or the linear hash [15] can be efficiently processed by SIMD. By contrast, our work focuses on probing a chained hash table, which suffers from more cache misses, and eliminating the cache misses. In addition, a permutation lookup table [29] is taken to introduce new elements in a vector, whereas we use the expand_load instruction to load new ones. Our algorithms are completely vectorized, unlike [28, 29, 7] which leave a scalar tail.

Divergence. Due to the lack of hardware supports like in GPUs [12, 3], CPUs have to manually handle the divergence of SIMD vectorization. The divergence within individual operators like build and probe can be avoided in the full vectorization [28]. This method can be extended to the execution of pipelines, which is named *partial consume* [22, 23]. Another strategy in [22, 23] named *consume everything*, buffers divergent tuples and defers their processing, which is similar to the residual vectorized states in this paper. However, "consume everything" introduces more nested *if* and *loop* statements in the buffering operator, making the processing logic more complex. In contrast, each residual vectorized state is owned by a divergent state of an FSM, and shared among a group of running instances. Besides, "partial consume" and "consume everything" also only consider the divergence between active vector lanes and inactive vector lanes, ignoring the divergence from active lanes. To this end, those two strategies cannot solve the divergence from general *if* and *loop* statements.

Prefetching in SIMD. Cache misses seriously limit the performance of memory-intensive applications. To solve this problem, prefetching is an effective way in scalar code using GP [6], SPP [6] or AMAC [20]. The latter in particular can handle irregular data access. With regards to vectorized code, some research efforts prefer adjusting the data layout to increase data locality and benefit from the hardware prefetching, and a few other studies preliminarily use the software prefetching. For example, a set of layouts are designed to reduce memory latency while traversing trees or graphs in [32, 18], but they cannot be used in other applications, like probing hash tables. Besides, SPP is also adopted to prefetch data in regularly traversing equal-height trees [18]. However, this way can only achieve at most $log_2(W)$ speedup, instead of *W*, where *W* is the number of lanes in a vector, so it wastes the high data parallelism in a vector. In addition, the software prefetching is taken to aid the sequential data accesses instead of random data accesses [7, 15], slightly reducing cache misses. Furthermore, ROF [26] links prefetching-optimized code and SIMD-optimized code in a pipeline, instead of exploiting prefetching to solve the cache misses in SIMD code.

8. CONCLUSION

We present the interleaved multi-vectorizing to break through the memory wall in SIMD vectorization. IMV is a new approach to fully utilize MLP and DLP on pointer-chasing applications with irregular and immediate memory accesses. IMV splits a program into states where the program encounters immediate memory access or control flow divergence, then IMV interleaves the execution of states from different running instances of the program to hide memory access latency. We also propose the residual vectorized states to solve the control flow divergence within each state so that there are no bubbles in vectorized execution. Experiments show IMV is up to 4.23X and 3.17X faster than pure scalar implementation and pure SIMD vectorization, because it can reduce cache misses, branch misses and computation overhead for an application at the same time. IMV not only works well in pointer-chasing applications, it can also be applied to the whole query processing. In the future, as discussed in Section 6, we will apply IMV to complex queries and attempt to implement IMV automation.

Acknowledgments

This research was supported by National Key Research & Development Program of China (No. 2018YFB1003400), and National Natural Science Foundation of China (No. 61772204 & 61732014). The authors would like to thank the anonymous reviewers and shepherd for their constructive comments and guidance.

9. REFERENCES

[1] IMV Source Code. https://github.com/fzhedu/db-imv, 2019.

[2] Intel Vtune TMAM. https://software.intel.com/en-us/vtune-amplifier-cookbook-top-down-microarchitecture-analysis-method, 2019.

[3] M. Alam, K. S. Perumalla, and P. Sanders. Novel parallel algorithms for fast multi-GPU-based generation of massive scale-free networks. *Data Science and Engineering*, 4(1):61–75, 2019.

[4] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[5] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.

[6] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. *ACM Trans. Database Syst.*, 32(3):17, 2007.

[7] X. Cheng, B. He, X. Du, and C. T. Lau. A study of main-memory hash joins on many-core processor: A case with Intel Knights Landing architecture. In *CIKM*, pages 657–666, 2017.

[8] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y. Chen, A. Baransi, S. Kumar, and P. Dubey. Efficient implementation of sorting on multi-core SIMD CPU architecture. *PVLDB*, 1(2):1313–1324, 2008.

[9] M. Dreseler, J. Kossmann, J. Frohnhofen, M. Uflacker, and H. Plattner. Fused table scans: Combining AVX-512 and JIT to double the performance of multi-predicate scans. In *ICDE*, pages 102–109, 2018.

[10] Z. Fang, Z. He, J. Chu, and C. Weng. SIMD accelerates the probe phase of star joins in main memory databases. In *DASFAA*, pages 476–480, 2019.

[11] T. Gubner and P. Boncz. Exploring query execution strategies for JIT, vectorization and SIMD. In *ADMS*, 2017.

[12] T. D. Han and T. S. Abdelrahman. Reducing branch divergence in GPU programs. In *GPGPU*, page 3, 2011.

[13] H. Inoue, M. Ohara, and K. Taura. Faster set intersection with SIMD instructions by reducing branch mispredictions. *PVLDB*, 8(3):293–304, 2014.

[14] H. Inoue and K. Taura. SIMD- and cache-friendly algorithm for sorting an array of structures. *PVLDB*, 8(11):1274–1285, 2015.

[15] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, 2015.

[16] C. Jonathan, U. F. Minhas, J. Hunter, J. J. Levandoski, and G. V. Nishanov. Exploiting coroutines to attack the "killer nanoseconds". *PVLDB*, 11(11):1702–1714, 2018.

[17] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB*, 11(13):2209–2222, 2018.

[18] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[19] V. Kiriansky, H. Xu, M. Rinard, and S. P. Amarasinghe. Cimple: instruction and memory level parallelism: a DSL for uncovering ILP and MLP. In *PACT*, pages 1–16, 2018.

[20] Y. O. Koçberber, B. Falsafi, and B. Grot. Asynchronous memory access chaining. *PVLDB*, 9(4):252–263, 2015.

[21] N. Kohout, S. Choi, D. Kim, and D. Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *PACT*, pages 268–279, 2001.

[22] H. Lang, A. Kipf, L. Passing, P. A. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. In *DaMoN*, pages 1–8, 2018.

[23] H. Lang, L. Passing, A. Kipf, P. Boncz, T. Neumann, and A. Kemper. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *The VLDB Journal*, 2019.

[24] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[25] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, 2000.

[26] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[27] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[28] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking SIMD vectorization for in-memory databases. In *SIGMOD*, pages 1493–1508, 2015.

[29] O. Polychroniou and K. A. Ross. Vectorized bloom filters for advanced SIMD processors. In *DaMoN*, pages 1–6, 2014.

[30] O. Polychroniou and K. A. Ross. Towards practical vectorized analytical query engines. In *DaMoN*, pages 1–7. ACM, 2019.

[31] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *PVLDB*, 11(2):230–242, 2017.

[32] B. Ren, G. Agrawal, J. R. Larus, T. Mytkowicz, T. Poutanen, and W. Schulte. SIMD parallelization of applications that traverse irregular data structures. In *CGO*, pages 1–10, 2013.

[33] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[34] W. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. In *Comp. Arch. News*, pages 20–24, 1995.

[35] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.