Mikhail Zarubin, Patrick Damme, Alexander Krause, Dirk Habich, Wolfgang Lehner

SIMD-MIMD cocktail in a hybrid memory glass: shaken, not stirred

Erstveröffentlichung in / First published in:

SYSTOR '21: Proceedings of the 14th ACM International Conference on Systems and Storage, Haifa 14. – 16.06.2021. ACM Digital Library. ISBN 978-1-4503-8398-1

DOI: https://doi.org/10.1145/3456727.3463782

SIMD-MIMD Cocktail in a Hybrid Memory Glass: Shaken, not Stirred

Mikhail Zarubin, Patrick Damme*, Alexander Krause, Dirk Habich, Wolfgang Lehner
TU Dresden, Database Systems Group, Dresden, Germany
firstname.lastname@tu-dresden.de

ABSTRACT

Hybrid memory systems consisting of DRAM and NVRAM offer a great opportunity for column-oriented data systems to persistently store and to efficiently process columnar data completely in main memory. While *vectorization (SIMD)* of query operators is state-of-the-art to increase the single-thread performance, it has to be combined with *thread-level parallelism (MIMD)* to satisfy growing needs for higher performance and scalability. However, it is not well investigated how such a SIMD-MIMD interplay could be leveraged *efficiently* in hybrid memory systems. On the one hand, we deliver an extensive experimental evaluation of typical workloads on columnar data in this paper. We reveal that the choice of the most performant SIMD version differs greatly for both memory types. Moreover, we show that the throughput of concurrent queries can be boosted (up to 2x) when combining various SIMD flavors in a multi-threaded execution. On the other hand, to enable that optimization, we propose an *adaptive SIMD-MIMD cocktail* approach incurring only a negligible runtime overhead.

CCS CONCEPTS

• **Information systems** → **Database query processing**; **Main memory engines**; **Phase change memory**; • **Computer systems organization** → **Single instruction, multiple data**.

KEYWORDS

hybrid memory; column store; SIMD; MIMD; optimization

*Now at Graz University of Technology and Know-Center GmbH, Austria.

ACM Reference Format:
Mikhail Zarubin, Patrick Damme*, Alexander Krause, Dirk Habich, Wolfgang Lehner. 2021. SIMD-MIMD Cocktail in a Hybrid Memory Glass: Shaken, not Stirred. In *The 14th ACM International Systems and Storage Conference (SYSTOR '21), June 14–16, 2021, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3456727.3463782

1 INTRODUCTION

Data analytical tools, e.g., interactive dashboards, are usually deployed on top of data systems. The tasks of those data systems are to persistently manage the data and to execute simple analytical queries over the data. As recently shown [62, 63], simple queries like `SELECT MIN(a), MAX(b) FROM r` are issued millions of times, e.g., to populate drop down fields in the dashboard. Similarly, queries like `SELECT SUM(a) FROM r WHERE b = const` are used to calculate tailored aggregates. These analytical queries typically access a small number of columns or attributes, but a high number of rows and are, thus, most efficiently processed using a columnar data organization [11, 12, 19, 58]. Nevertheless, the multitude of these queries demand two major performance requirements of columnar data systems: (i) a high query throughput, since analyses are performed by many users concurrently, and (ii) a low query latency, since analyses are expected to be interactive. Requirement (i) is commonly addressed by leveraging the well-known *Multiple Instruction Multiple Data (MIMD)* parallel paradigm, also known as thread-level parallelism. Here, each query is processed by an individual thread. Addressing requirement (ii) typically involves the *Single Instruction Multiple Data (SIMD)* parallel paradigm. Here, a single SIMD instruction processes multiple data elements at once, thereby increasing the single-thread performance. Both techniques have been available in x86-processors for many years, and their combination is a logical necessity to address the need of an overall high throughput with low query latencies across the board.

To satisfy this demand, symmetric or scale-up multiprocessor systems (SMPs) are a promising hardware foundation. SMPs are composed by multiple processors (also called nodes or sockets), each consisting of the same architecture,

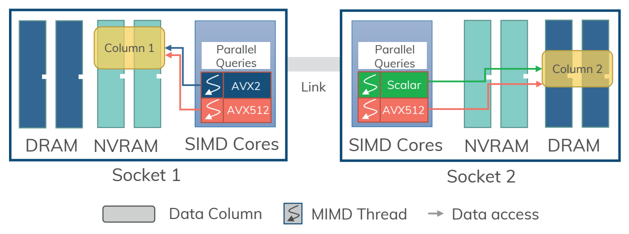

Figure 1: *SIMD-MIMD cocktail* approach for the concurrent and vectorized execution of analytical queries in hybrid memory systems.

e.g., a multi-core processor where each core features different SIMD instruction set extensions such as SSE, AVX2, or AVX-512 on Intel processors and all multiprocessors share a common and huge main memory space. An observable hardware trend of recent years is the increase of both, the number of cores per socket and the size of vector registers, thereby facilitating both MIMD and SIMD parallelism. Moreover, these SMPs are more and more shifting towards *hybrid memory systems*, that consist of both volatile DRAM and persistent NVRAM, as illustrated in Figure 1. This offers a great additional opportunity for columnar data systems (column-stores) to persistently store and to efficiently process huge amounts of columnar data exclusively in main memory without touching any slow block-accessible non-volatile medium. However, combining MIMD and SIMD is impaired by physical limitations of the hardware. Especially thermal constraints and on-chip power supply can decrease the actually available resources, e.g., through a reduction of the clock frequency, depending on the employed SIMD extension and the number of used threads [24, 27].

Our contributions and outline. To reach the best query performance in column-stores on scale-up hybrid memory systems, we propose a *SIMD-MIMD cocktail* approach for the concurrent *and* vectorized execution of a set of simple, yet important analytical queries in this paper. In detail, our contributions and paper outline can be summarized as follows:

(1) In Section 2, we provide an in-depth description of our underlying system model.
(2) Then, we develop our vision on how to combine the potential performance gains of SIMD and MIMD. For this, we provide an experimental analysis of typical analytical query workloads and inspect the effect of the simultaneous usage of different SIMD instruction set extensions in a multi-threaded environment.
(3) In Section 4, we show how to effectively select a suitable SIMD-MIMD cocktail for a particular query workload. Our evaluation subsequently confirms the applicability of our model for queries on both DRAM and NVRAM memory types.

Finally, we present related work in Section 5 and we conclude the paper with a short summary in Section 6.

2 SYSTEM MODEL

In this section, we cover all relevant aspects of our system model by (i) presenting the data and processing model, (ii) describing data placement options for our target hardware, and (iii) introducing the parallel processing opportunities.

2.1 Data and Processing Model

As commonly used by state-of-the-art analytical data systems, we employ a columnar data representation for base data [1, 2, 21, 31, 36, 55, 73]. Here, relational data is maintained using the decomposition storage model (DSM) [17], where each column of a table is stored separately as a fixed-width dense array [1]. To allow easy reconstruction of the tuples of a relational table, each column record is stored in the same (array) position across all columns of a table [1]. Column-stores typically support a fixed set of basic data types, including integers, fixed-, or floating-point numbers, and strings. For fixed-width data types (e.g., integer, fixed-, and floating-point), column stores utilize basic arrays of the respective type for the values of a column. However, floating-point numbers are usually mapped to integers [1] as well. Variable-width data types like strings are generally dictionary encoded and represented as integers, which enables their storage into fixed-width columns, too [1, 8]. In the simplest case, a dictionary consists of the distinct values of a column, sorted by frequency, and each value is represented as its integer position on the dictionary [1]. Consequently, all base columns consist of a sequence of fixed-width integers.

For an efficient processing of these sequences, the column-at-a-time model is heavily applied in these systems [1, 10]. Here, an SQL query is translated into a query execution plan (QEP) consisting of multiple operators. Typical query operators include `select`, `project`, `aggregate`, `join`, `group-by`, and `set`-operations featuring a mixture of sequential and random memory accesses for reads and writes. Each operator consumes one or two input columns and produces an output column called intermediate. The column-at-a-time processing model explicitly materializes these intermediates, because each operator within a QEP is evaluated to completion over its entire input, before subsequent data-dependent operators are invoked. Such intermediates are volatile columns and thrown away during or right after the query execution. Thus, we distinguish between persistent *base data* and ephemeral *intermediates* in our system model.

2.2 Data Placement

Column-store systems completely keep data in DRAM for efficiency, but a consistent copy of the base data has to be stored on a non-volatile medium for persistency and fault-tolerance [1, 2, 10, 21, 36, 55, 73]. Traditional persistent storage mediums like HDDs or SSDs are employed for durability, but suffer from a slow and block-addressable access. To overcome that, NVRAM—also known as persistent memory—enables a pure in-memory behavior with a byte-addressable fashion, while preserving the durability property of traditional storage solutions. Its performance is approaching that of DRAM, especially for a sequential access pattern [28, 66, 71]. However, as a drawback compared to volatile DRAM, NVRAM has lower endurance and maximum bandwidth limits. Thus, only base columns are stored in NVRAM, while volatile intermediates are usually materialized in DRAM. Consequently, NVRAM is a good candidate for the extension of fast and byte-addressable capacity while simultaneously serving as a replacement for traditional secondary storage elements, that store the primary copy of the base data [4, 5, 33, 41, 44, 70].

Hybrid memory systems featuring DRAM and NVRAM are becoming a standard in the field of high performance server architectures [49]. In such systems, each local memory domain consists of DRAM and NVRAM at the same hierarchy level [22], which is also illustrated in Figure 1. Thus, applications require only insignificant changes compared to DRAM-backed processing. The respective functionality, like memory access, allocation, and persistent flushes, is provided by specialized convenience tool-kits, whereas the Persistent Memory Development Kit (PMDK) [56] by Intel is the most well-known. Furthermore, hybrid memory systems can also be scaled up by deploying them in a Non-uniform Memory Access (NUMA) environment. NUMA systems consist of multiple physically separated processors, which each feature their own local memory hardware resources [47]. In fact, Figure 1 depicts a two-socket NUMA system. Since NUMA-oriented scale-up systems essentially behave like distributed systems, but feature a faster communication due to cache coherency facilities and the close proximity of processors, the near-memory processing paradigm (NMP) is state-of-the-art on such platforms [30, 34, 46]. The performance implications of NUMA-effects are well known [30, 34, 46] and thus, we focus on memory-local (either DRAM or NVRAM) execution.

2.3 Parallel Processing Opportunities

Scale-up hybrid memory systems consist of numerous cores offering two parallel processing paradigms: *Single Instruction Multiple Data (SIMD)* and *Multiple Instruction Multiple Data (MIMD)*. While SIMD applies a single instruction to a vector of multiple data elements, MIMD applies multiple threads over different or the same data at the same time.

SIMD—also called vectorization—is a state-of-the-art optimization technique in columnar data systems and is typically applied to isolated query operators [2, 50, 72]. Many vectorized implementations for joins [6, 9] and sorting [51] have been proposed. Moreover, linear access operators such as scans [64] and integer compression techniques [2, 18, 37] are well-investigated. In the past years, hardware vendors have regularly introduced new SIMD instruction set extensions operating on increasingly wider registers. For instance, Intel's Advanced Vector Extension (AVX2) operates on 256-bit vector registers and Intel's AVX-512 uses even 512-bit vectors. Wider vector registers allow processing more data elements at once. For example, an Intel SSE 128-bit vector register can store two 64-bit data elements, while AVX2 and AVX-512 can store twice or four times the amount, respectively. Recently, Ungethüm et al. [59] introduced a specific SIMD abstraction layer called *Template Vector Library (TVL)* for column-stores to tackle the SIMD diversity in a unified way. On the one hand, the *TVL* offers hardware-oblivious vector primitives. On the other hand, the *TVL* also provides an extensible set of hardware-conscious implementations for the hardware-oblivious primitives. We are using that approach to realize hardware-oblivious vectorized query operators based on the provided vector primitives which can be easily mapped to specific hardware-conscious implementations.

MIMD is a heavily used optimization technique in column-stores as well, whereby two approaches can be distinguished. On the one hand, MIMD is used to realize a *data-partitioned intra-operator* parallelism. Here, every column is partitioned into data chunks that are exclusively processed by one operator. More precisely, every operator is parallelized through a set of spawned sub-operators [52]. These sub-operators are mapped to designated threads and every thread is assigned to a specific column partition. This OpenMP-like processing style is typically used for operators with a sequential memory access pattern and does not require any sophisticated controlling mechanism, since there are mostly no data or control flow dependencies between processed partitions. On the other hand, MIMD is utilized for a *multi-threaded inter-query* parallelism. This approach maps one query to one thread at any point in time and may also employ *snapshot isolation (SI)* [13] on the software level, to allow for a high degree of parallelism. In SI, queries atomically take a snapshot of all data columns they need to access during their prologue phase. From there on, the queries do not see any changes made to those columns after that point in time. To leverage potential caching of shared data in this case, it is common to batch a number of queries that are touching the same columns [23, 38, 53]. Then, the execution of this query batch is triggered at once. Usually, all involved query operators are vectorized using the newest SIMD instruction set extension such as AVX-512 [52]. However, this SIMD-MIMD interplay is sub-optimal for the inter-query parallelism approach as we are going to show in the next section.

.

3 ANALYZING SIMD-MIMD INTERPLAY

After presenting our column-store system model, this section is devoted to the experimental exploration of the SIMD-MIMD interplay for concurrent query execution in a hybrid memory system. Based on this experimental analysis, we subsequently propose an approach to select a suitable SIMD-MIMD cocktail for a particular query workload in Section 4.

3.1 Evaluation Setup

Our evaluation platform is a two-socket hybrid memory NUMA system equipped with Intel Xeon Platinum 8276L (Scalable Cascade Lake family) processors, 384 GiB DDR4 DRAM, and 1.5 TiB Intel Optane DC Persistent Memory. Only CPUs of a single socket are used, i.e., 18 physical cores (36 with HyperThreading). Besides scalar processing, each core provides the following Intel SIMD instruction set extensions: SSE with 128-bit, AVX2 with 256-bit, and AVX-512 with 512-bit vector registers. The server runs Fedora 27 with kernel version 5.4.45 (CPU governor is set to "performance"), and `gcc 8.3.0` with -O3 flag was used for compilation. The NVRAM chunks are allocated using memory mapped files (PMDK-style) on XFS file system.

For columns in DRAM as well as NVRAM, we use 100 M 64-bit integer values (763MiB) uniformly drawn from the interval [1, 1M]. As already mentioned in the introduction, we focus on simple analytical queries. To execute these queries, the following columnar query operators are required:

The **aggregate-operator** is a read-intensive operator performing a certain cumulative operation (e.g., summation) over the input column following a sequential access pattern, while only a single element is written to the output column.

The **select-operator** performs a scan over the single input column and outputs the positions of the data elements fulfilling a certain filter predicate. Depending on the amount of selected elements, the operator is either read-dominated or read-write balanced.

The **project-operator** is used to transfer the result of a selection on one column X to another column Y by using the positions in column X to gather the corresponding values from column Y. Thus, this operator typically mixes random reads with sequential writes.

As introduced in Section 2, the output of each operator is an intermediate result and, thus, written to DRAM. The input columns could be from DRAM or NVRAM. Aside from these read-intensive operators, we also investigate the *write-only* **append-operator**. As commonly done in big data and data warehouse applications, new values are added to the end of the columns by the append-operator [14].

We implemented all operators using the specific SIMD abstraction library *TVL* for column-stores [59][1]. Based on

[1] https://github.com/MorphStore/TVLLib

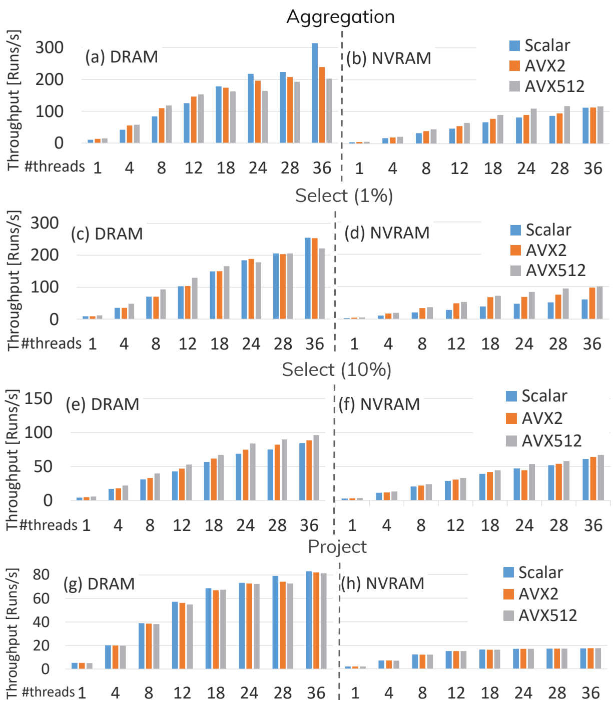

Figure 2: The performance of read-intensive operators for various SIMD options and two memory types.

that, we are able to automatically derive variants for the different Intel SIMD extensions as well as a scalar variant. In our experiments, we investigate both the SIMD and scalar operator variants. Finally, we report performances in terms of *runs per second*, i.e., how many times the particular operator was executed by all threads within 1 second (averaged through a 1 minute execution period).

3.2 Impact of Vectorization

To analyze the interplay of SIMD and MIMD, we investigate how operators behave in a concurrent setting, when they are vectorized with different SIMD extensions. In detail, we consider scalar execution, AVX2, and AVX-512 extensions. The results for SSE are omitted, as they mostly exhibit a similar behavior as scalar or AVX2 vectorization. In addition to the different SIMD variants, we also vary the number of cores or the number of operators that are executed simultaneously. In particular, we examine the typical use case where all concurrent operators access a single shared column.

Aggregate-operator: The throughput of this operator is illustrated in Figures 2-(a,b) for base columns stored in DRAM and NVRAM, respectively. Our first observation is that the execution behavior differs significantly for the two memory types. For NVRAM, we observe the expected behavior, that larger registers imply higher performance, even with

.

increasing thread count. When all cores are active, we can no longer observe a significant difference between SIMD variants, most likely through memory bandwidth saturation. The DRAM case, while showing significantly higher throughput (reflecting the higher read bandwidth limits), yields surprising results. Primarily, we see that the NVRAM-like large register's domination is only visible until a certain concurrency level (12 threads). Afterwards, the efficiency leadership is taken over by smaller counterparts, e.g., by scalar starting from 18 threads. Apart from this, we observe that performance gains are also possible until full CPU occupancy of 36 threads. We assume that these effects are induced by the caching of large registers and the resulting contention, which could not be reached in case of NVRAM due to differences between persistent and volatile memory controllers [28] and slower cache replacement/trashing.

`Select`-operator: Our next operator under test is the `selection`. Here, we distinguish two degrees of selectivity (1 % and 10 % of qualifying elements) as they obviously impact the resulting memory access pattern. The first case is shown in Figures 2-(c,d). Since a sequential access pattern is dominating here, we can mostly confirm the observations made for `aggregation`. The most notable difference is the decreased level of performance variations between SIMD variants on DRAM, while the opposite is observed for NVRAM. AVX-512 loses its domination on volatile memory later, just starting from 24 threads onwards, while the scalar implementation is not able to significantly outperform AVX2 at all. The situation changes with the increase of the selectivity percentage.

Figures 2-(e,f) demonstrate the case of 10%. Now, more data has to be written, which changes the pressure on the memory controller. This becomes even more severe in multi-threaded scenarios. As a result, we see a general performance drop for both memory types, compared to their 1% counterparts. Furthermore, the impact of the employed SIMD version to vectorize the operator becomes less significant, however large registers are still preferable in most cases. Lastly, we observe that a slight performance increase is still reachable until full CPU occupancy for both mediums.

`Project`-operator: The last read-balanced operator in our analysis is `projection`. Figures 2-(g,h) demonstrate the case of data/positions size ratio equaling to 10%, while positions are unsorted and uniformly distributed. In such a scenario, the output column would contain 10 times more elements than the input column. The key difference from a memory access pattern point of view, compared to the previous cases, lies in the randomness of reads when extracting the targeted elements. As already reported by previous research [52], such stochastic data accesses can diminish the performance advantage of using large registers. Indeed, we

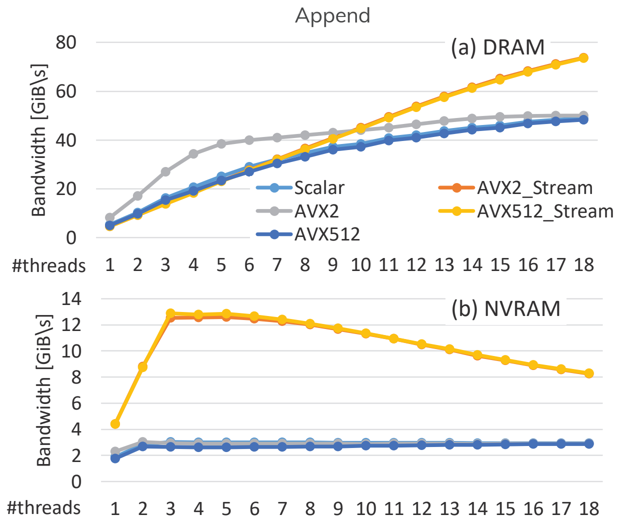

Figure 3: The performance of the append-operator for various SIMD options and both memory types.

confirm such behavior for both memory types and all concurrency levels. While insignificant advantages of smaller registers could still be detected on DRAM, there is virtually no difference between the SIMD versions on NVRAM. However, the general performance of persistent memory execution drops reflecting its pure random read latency and bandwidth limits. These limits also determine the thread counts after which performance gains are not feasible any more (e.g., from 18 threads for NVRAM).

`Append`-operator: As already mentioned, with regards to data modifications, we focus exclusively on the append-operator. The respective experiments are illustrated by Figures 3-(a,b). Here, we propagate synthetically generated data from CPU registers to be *sequentially* stored at the end of the targeted columns, whereby each thread appends to an individual column. Neither cache leveraging nor `memcpy()` operations are involved. Our first observation is that there is a huge difference between the two memory types, not only in terms of performance. While DRAM prefers AVX2 as the fastest write mechanism for a moderate concurrency level, there is almost no difference between SIMD flavors on NVRAM. However, what really matters for NVRAM is the streaming style [26] of stores, as they are able to deliver up to 5x better throughput (e.g., for a thread count of 3), compared to the non-streaming counterparts. Another crucial result of our experiments is that there are break-even points where one implementation hands over its leadership to another. For instance, from 10 threads onward, AVX-512 with streaming store is the fastest for DRAM. Moreover, there are most-performing points after which the performance starts

Final edited form was published in "SYSTOR '21: Proceedings of the 14th ACM International Conference on Systems and Storage, Haifa, 2021. ACM Digital Library. ISBN 978-1-4503-8398-1
https://doi.org/10.1145/3456727.3463782

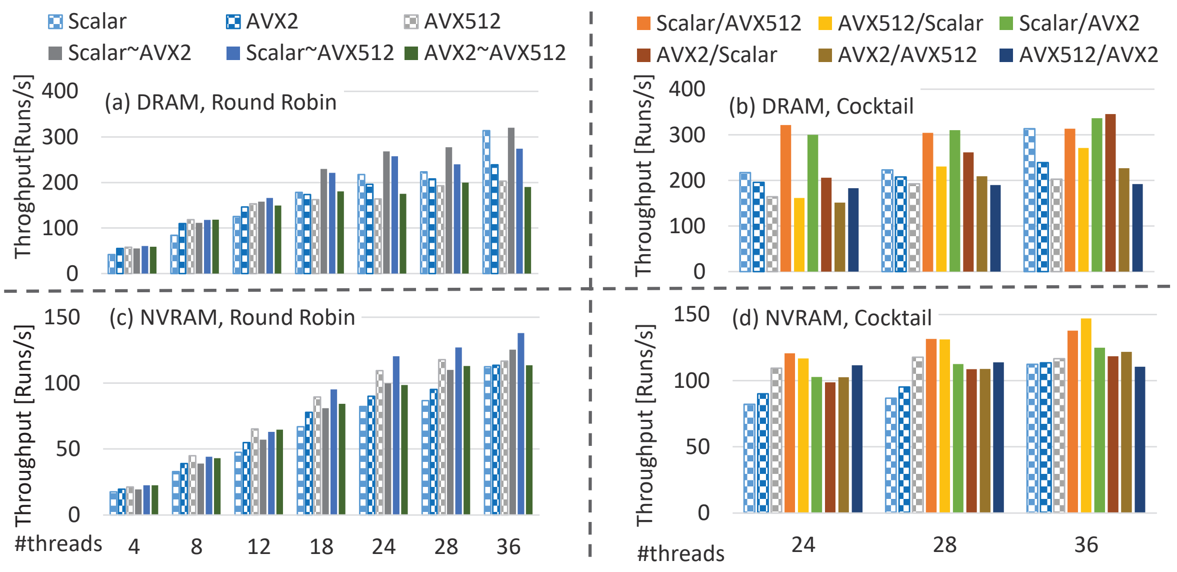

Figure 4: The performance of `aggregate`-operators for various homogeneous and mixed SIMD options.

to drop. For example, from 4 threads onward, the performance of streaming stores for NVRAM deteriorates.

3.3 SIMD-heterogeneous MIMD Execution

Since our system model assumes the use of MIMD in form of an inter-query parallelism (cf. Section 2.3), it is possible to use *distinct* SIMD extensions (or scalar processing) per thread, e.g., simultaneously run scalar and AVX-512 schemes, possibly on the same data columns. This combination of heterogeneous SIMD-MIMD parallelization is worth investigating, as the usage of AVX-512 registers for multiple threads leads to a reduction of the CPU's clock frequency and, thus, diminished performance gains. Hence, this section is devoted to the respective experimental examination. To the best of our knowledge, mixing SIMD extensions and scalar processing was not investigated before.

The evaluation design space involves *all distinct combinations* of available SIMD extensions (or scalar processing) per thread, multiplied by the number of used threads. For illustration purposes, we limit our analysis to a selection of options, combining two SIMD flavors at a time. We use the following abbreviations for such *shaking* mechanisms:

SIMD1~SIMD2: means that the employed cores are evenly (e.g., one-by-one) distributed between two SIMD options in a *round robin* (RR) or *alternating* fashion.

SIMD1/SIMD2 is what we call the sliced *cocktail* style. It devotes the first half of all employed CPUs, i.e., CPU1 through CPU18 or all *physical* cores, to the SIMD1 vectorization flavor (reflecting the "bottom" slice). If more than 18 cores are used for the query execution, we fill the SIMD2 flavor on top for these additional resources. This allows us to vary the *ratio* between both flavors.

`Aggregate`-operator: Figures 4-(a-d) show the results for both *RR* and *cocktail* style on both DRAM and NVRAM. We omit thread counts below 24 for the latter, as they perform identically to the previously shown pure SIMD variants. Most importantly, we reveal that the homogeneous SIMD alternatives (depicted as pattern filled bars) can be outperformed by our suggested combinations for all considered experimental setups. However, the particular throughput improvement varies depending on the shaking mechanism (i.e., RR or cocktail), concurrency level and memory class. For instance, on DRAM, for an intermediate level of CPU occupancy (18 threads) the round robin "Scalar~AVX2" shaking yields a 29% increase compared to the best (Scalar) homogeneous approach, while NVRAM favors the "Scalar~AVX512" scheme with 5% speedup (over "AVX512") for the same case. For higher degrees of concurrency, our cocktail shaking mechanism shows a surprising behavior. On DRAM, for 24 concurrent threads, "Scalar/AVX512" is able to double the performance of the pure AVX-512 implementation. Interestingly, the opposite approach "AVX512/Scalar" is by far not the best on volatile memory, while it is the superior scheme for persistent memory—delivering a 10% improvement for 36 threads.

`Select`-operator: Comparable behavior is detected for the `select`-operator, but only for small selectivities, i.e., only a few percent. While varying the selectivity on NVRAM, we observe that no throughput increase through shaking SIMD flavors was possible for any selectivity higher than 1%. We assume that this is due to lower performance of writes that

Final edited form was published in "SYSTOR '21: Proceedings of the 14th ACM International Conference on Systems and Storage, Haifa, 2021. ACM Digital Library. ISBN 978-1-4503-8398-1
https://doi.org/10.1145/3456727.3463782

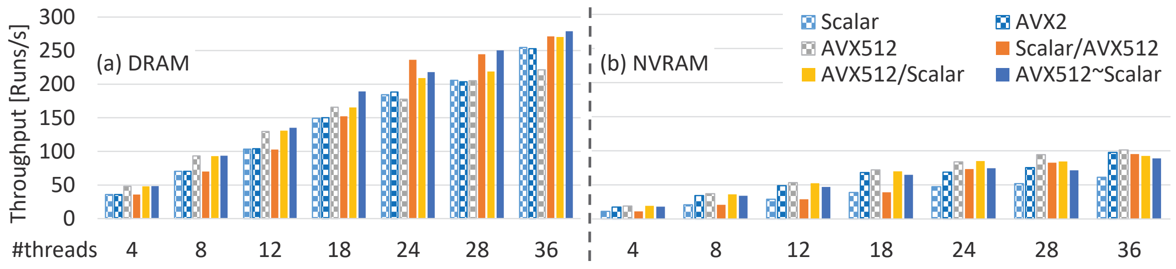

Figure 5: Performance of select-operators (1%) for various homogeneous and mixed SIMD options.

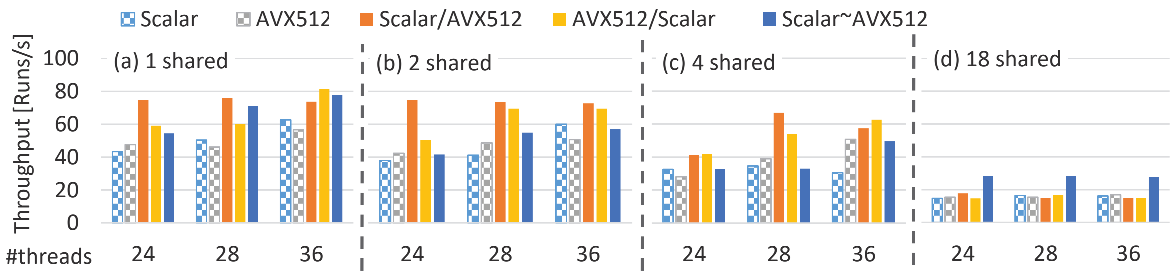

Figure 6: DRAM performance of aggregate-operators for various SIMD options and different # shared columns.

are proportional to the selectivity degree. Thus, Figure 5 depicts our measurements for 1% selectivity on both memory types. Though the mixed performance gains are lower than for the aggregation-operator, they still provide significant improvement, e.g., 25% for "Scalar/AVX512" compared to the best homogeneous option on DRAM for 24 threads.

Impact of number of shared columns: These facts lead to the assumption that such unexpected performance gains most likely depend on the way how various SIMD extensions interact with the caching subsystem, which can in turn favor sophisticatedly downclocked physical or hyper cores. Thus, we investigate the actual impact of caching via experiments by using different numbers of very large shared columns. The aggregation-operator on volatile memory is depicted by Figures 6-(a-d) for a single, two, four, and eighteen shared columns, respectively. The actual data size was tripled, i.e., increased to 2.3 GiB per column, to reduce spatial locality influence. The previously detected behavior of SIMD mixtures (e.g., possible superior performance) combined with *sequential access* operators is preserved for up to 18 shared columns per 36 queries, i.e., 1 column is shared by 2 threads. However, the level of throughput diversity between mixed schemes is decreasing with the number of shared columns. With 18 columns, only the round robin scheme considerably outperforms the homogeneous vectorization and, thus, yields a performance increase. The data access pattern exhibits crucial importance as well, as for random-read or heavily read-write mixed operators (e.g., project or select with a high degree of selectivity), the actual mixed performance increase tends to disappear even for the single shared data scenario.

3.4 Lessons Learned

From our experimental analysis of typical data intensive workloads, we draw the following important conclusions: (1) Given the assumptions of our system model, the employment of SIMD parallelism can significantly improve the performance compared to a scalar execution. (2) However, it is important to carefully select among the available SIMD options (including scalar processing) to reach the highest performance. A naïve strategy of always selecting the largest registers, i.e., AVX-512, may be even harmful compared to the scalar execution. (3) Furthermore, even within one instruction set, different ways to store data (e.g., streaming) can have a significant impact. (4) The best-performing SIMD option depends on the level of concurrency, the memory type, and the operator (or access pattern). (5) Our newly discovered effects of SIMD-cocktails offer a great opportunity for performance optimization, and in the following section, we present a strategy to leverage them effectively.

4 OPTIMIZING SIMD-MIMD INTERPLAY

As described in our system model in Section 2, batching of queries in a multi-threading environment is a common optimization technique to benefit from caching effects. This batch of queries is executed at once and all operators are typically vectorized using one specific SIMD extension such as

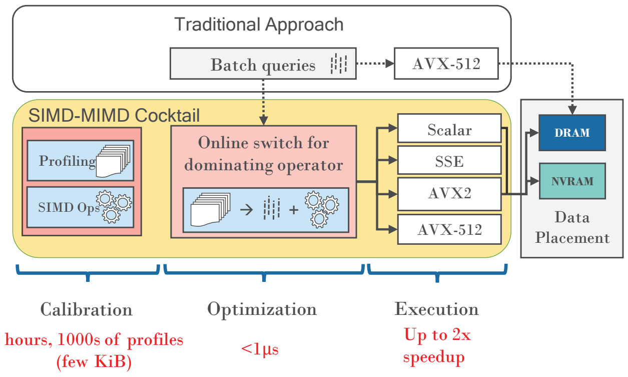

Figure 7: Deployment model (yellow box) of our adaptive SIMD-MIMD vectorization cocktail.

AVX-512 in DRAM as shown in Figure 7. However, as clearly shown in the previous section, the choice of the optimal SIMD version applied for operator vectorization depends on various factors such as the degree of MIMD parallelism, memory type, access pattern and level of shared (and, therefore, potentially cached) data processing. In particular, Figures 3, 4, and 5 indicate that some vectorization schemes exhibit a significantly higher performance compared to others for certain fixed thread counts. Based on those observations, we now propose and evaluate an optimization design for an online decision mechanism as shown in Figure 7 to shake the best fitting cocktail for the current conditions at run-time.

4.1 Optimization Design

Our proposed online decision mechanism extends the existing query optimizer of a database system. The main task of a query optimizer is to translate a descriptive SQL query into an efficient query execution plan consisting of several operators. To shake the best fitting cocktail for the operators under the current conditions, our approach consists of the following three components:

Profiles. The profiles characterize the behavior of particular operators (or access patterns). Such *platform-dependent* profiles are generated *once* at deployment time and provide information about the performance of the individual vectorization schemes (either homo- or heterogeneous) in the context of various concurrency levels and data set configurations. The profiles build the foundation for the adaptive optimization mechanism. Essentially, one profile per combination <operator> x <medium> x <#threads> x <#shared columns> is retrieved. For illustrative purposes, in the following we stay with the measurements of Section 3, which provide us with profile information for our test system. We use a table format as shown in Table 1, which reflects the case of 24 threads in Figure 4-(b). Thus, a single profile consists

Table 1: An example profile for <aggregate> on <NVRAM> @ <24> threads over <1> shared column.

SIMD Version	#Core Begin	#Core End	Round Robin
Scalar	1	18	No
AVX2	–	–	–
AVX-512	19	24	No

of several rows corresponding to the available SIMD extensions in the system, while each row specifies the enumeration interval of cores that has to run accordingly vectorized queries/operators to produce an optimal cocktail. The example in Table 1 tells us we should use scalar processing for threads 1–18 and AVX-512 for threads 19–24. Furthermore, the profile indicates that the cocktail-style is to be favored (for round robin, the corresponding cell would contain the "Yes" indicator). This format allows for profiling of multiple SIMD versions (e.g., more than two) used in the cocktail.

Model. As high performance is among the highest priorities of hybrid memory database systems, the switching algorithm is required to be very lightweight to keep its overhead as low as possible. Due to this essential requirement, we adopted a small lookup formula/function as our model for the selection algorithm. This formula is derived based on the measurements obtained at deployment time. Essentially it returns the profile best fitting to the current conditions, and is defined as: `Profile = F(<memory>, <operator>, <#threads>, <#shared columns>)`.

Online Switching. The actual online switching component leverages the information about the current workload (or batch) including queries, memory type and the data set configuration to calculate the aforementioned formula and navigate to the recommended profile. This step imposes only a few microseconds of fixed run-time overhead and thanks to the batching, we do not have to carry out online monitoring. The batches are executed one after the other and contain all necessary information. Subsequently, that profile is used to find out the optimal SIMD-configuration of vectorized operators to be executed by MIMD-parallel queries. Here, by workload information we understand the dominating operator (in terms of run-time) within the currently executed query and data set, which provides information about the number and size of shared data columns and the respective memory type. The complete procedure can be executed in user-space and is possible with user-space knowledge, yet it imposes only negligible overhead due to its simplicity and is usually completely amortized by the improvements of the query execution run-time.

Final edited form was published in "SYSTOR '21: Proceedings of the 14th ACM International Conference on Systems and Storage, Haifa, 2021. ACM Digital Library. ISBN 978-1-4503-8398-1
https://doi.org/10.1145/3456727.3463782

4.2 Implementation

We implemented a proof-of-concept of our switching approach using the columnar query engine *MorphStore* [19, 25]. The schematic view of our implementation is shown in Figure 7. The advantage of using *MorphStore* is that its operators are implemented in a hardware-oblivious way using the vector abstraction library TVL [59]. Thus, each operator implementation can automatically be specialized to different SIMD extensions, which is required for our overall approach. In our proof-of-concept, all alternative vectorized operators that can be possibly chosen are pre-compiled as isolated functions that can be called by our online decision mechanism. As illustrated in Figure 7, we batch concurrent queries accessing the same base columns [23, 38, 53].

Such a batch determines the characteristics of the workload (the concurrency level, memory type, and the dominating access pattern (i.e., operator)) that are given as parameters or can be extracted during query compilation phase without any significant overhead. We generated a decision formula as our switching model based on the profile information. Obviously, this model is *platform-dependent* and needs to be calibrated for different hardware platforms. However, this calibration has to be done only once per platform.

4.3 Evaluation

To show the efficiency and applicability of our optimization, we now present selective experimental results and discuss the deployment costs. The evaluation setup is the same as in Section 3, but we target more complex scenarios, not just single operators. Thus, we analyze the behavior of adaptive SIMD-conscious vectorization exemplified by three selected queries similar to those we mentioned in Section 1:

(1) `SELECT SUM(a) FROM r`

This query resembles the basic aggregation case involving a single shared column analyzed in Section 3.

(2) `SELECT SUM(a), SUM(b), SUM(c), SUM(d) FROM r`

Here, we extended the aggregated data set to four separate equally sized columns being scanned in a sequence in accordance with the column-at-a-time processing model.

(3) `SELECT SUM(a) FROM r WHERE x < const`

This query consists of three operators: selection, projection and aggregation. Both selection and projection (of different instances of this query) access shared base data, however, we ensured the superior domination of selection by using a selectivity of 1% on a large base column.

The resulting performance of the optimization mechanism is reflected by Figure 8 for both mediums. Here, for query (1) our decision mechanism is always able to select the best SIMD vectorization scheme among the ones we consider (Figures 8-(a,b)). This is, however, expected as the query execution can be exactly mapped to the aggregation profile

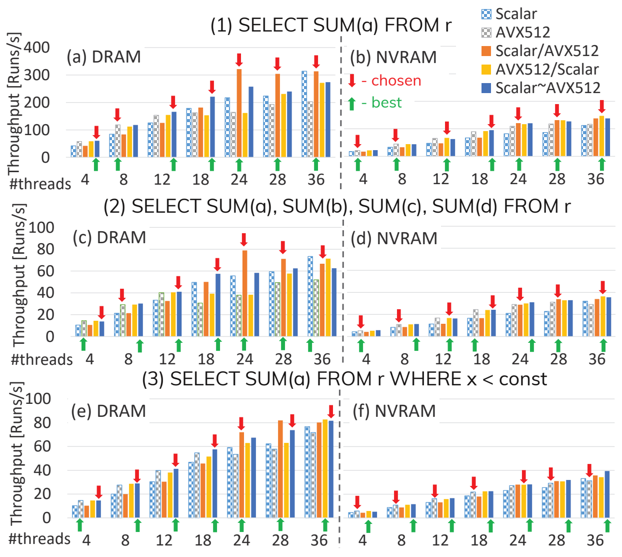

Figure 8: The performance of test queries indicating best measured and our selected SIMD combination.

and demonstrates the respective performance. The situation slightly changes with regard to query (2) which features its specific data set, while preserving the absolute domination of the `aggregation`-operator. According to our optimization assumption, the switching model again deploys the profile of elementary aggregation. The respective decisions result in a strong correlation with the best reachable performance on both DRAM and NVRAM (Figure 8-(c,d)). Although the actual best measured strategy is not always selected by the model (e.g., for thread count 4, 8 and 36 on DRAM), the respective performance losses compared to the optimum do not exceed a few percents. Finally, the behavior of query (3) is presented by Figures 8-(e,f) for volatile and persistent memory, respectively. As mentioned before, the selection part of this data processing task dominates within its runtime and, therefore, our optimization model deploys the profile of elementary selection (i.e., extracted from Figures 2-(c,d)), the remaining operators follow their default SIMD policies. The DRAM case demonstrates a high hit rate with the best measured option (only thread counts 1, 28 and 36 are mispredicted) with a worst-case loss of 10% compared to the best setting (e.g., for 28 threads). Nevertheless, the approach chosen in this case still outperforms the best respective basic scheme (Scalar) by 11%. The NVRAM-backed adaptive selection faces much worse correlation with the optimal measured scheme. However, this is not an issue for this scenario as all involved SIMD schemes demonstrate roughly similar throughput here, due to the slower nature of persistent memory. Thus, the under-gained performance does not exceed 7% (except for a single outlier at 36 threads).

.

Deployment Costs. The deployment of our proposed optimization imposes the following costs: (i) implementation of alternatively SIMD vectorized operators (though possibly automated using the TVL, cf. Section 2.3); (ii) one-time profile calibration overhead at system deployment phase—several hours and a few KiB of memory space to store thousands of measured profiles; (iii) a few microseconds of runtime overhead spent in the model for online switching.

Lessons Learned. Based on our evaluation, we conclude that our adaptive SIMD-conscious vectorization approach is applicable and useful. In nearly all examined cases, it is able to suggest either the optimal or an only slightly sub-optimal solution. While the performance delivered that way is able to considerably outpace the static homogeneous strategies (e.g., always using AVX-512), the run-time overhead is negligible. It is important to note that our approach remains useful even if only basic SIMD vectorization schemes are explicitly allowed as, in principle, it will just limit our decision making algorithms to the corresponding profiles (with the goal to select the best among the homogeneous options).

5 RELATED WORK

We have already discussed the basics of column-store systems, data placement in hybrid memory architectures as well as data-level parallelism based on SIMD and thread-level parallelism based on MIMD for column-stores in Section 2. Now, we focus on works related to the SIMD-MIMD cocktail and hybrid-memory systems in general.

Vectorization is widely used in column-stores [19, 36, 39, 55], with some query engines being even fully vectorized [52]. However, these systems typically focus on a single SIMD extension (usually AVX-512), i.e., they do not decide at run-time which extension to use or whether to mix extensions. In contrast to that, we found out that having multiple SIMD variants of the same columnar operator available and mixing them appropriately often improves query performance in multi-threaded settings.

Additionally, there are a couple of system-level works addressing the impact of AVX-512 on the processor's clock frequency. Gottschlag et at. [24] observe that AVX-512 code can slow down scalar code running shortly afterwards or concurrently on another hyper-core of the same physical core. They propose separating threads employing AVX-512 from those executing only scalar instructions by scheduling them on different physical cores to limit the slow-down incurred by AVX-512 on concurrent scalar code. Kumar et al. [35] propose to de-vectorize short vectorized code sections using JIT compilation techniques to avoid the negative impact on scalar code. Unlike in these works, in the setting we assume, the system is able to decide *if* a *pre-compiled* vectorized or scalar operator variant should be executed. Nevertheless, our SIMD-MIMD cocktails can also result in a physical separation of AVX-512 and scalar threads.

Regarding thread-level parallelism (MIMD), our work is related to scan-sharing [54]. Here, the idea is to make queries scanning the same data run at the same time to reduce the number of cache misses. Our approach of mixing SIMD extensions also requires multiple queries on the same base data at once. Nevertheless, we could show that even a low number of concurrent queries suffices to render the heterogeneous use of SIMD extensions beneficial.

Finally, with the advent of NVRAM, hybrid memory systems have gained importance in recent years. A number of persistent programming tool-kits, memory allocators and file systems [3, 7, 20, 29, 40, 56, 57, 65, 68, 69] have been developed. These components allow convenient deployment of NVRAM in a variety of applications, ranging from scientific HPC systems [48] to big data infrastructure, via the development of persistent memory-centric algorithms and data structures [15, 16, 32, 42, 43, 60, 61, 67]. Moreover, there is a number of hybrid memory data systems such as SAP HANA [4], SOFORT [41, 44, 45], FOEDUS [33], and Peloton [5]. We believe that our *cocktail* approach of mixing SIMD extensions is a suitable means to accelerate the access to NVRAM in these systems, which is especially crucial given the lower bandwidth of NVRAM compared to DRAM.

6 CONCLUSION

To achieve highest performance in analytics, both data-level parallelism using SIMD and thread-level parallelism using MIMD are mandatory in scale-up hybrid memory databases. In this paper, we performed a thorough evaluation and analysis of typical SIMD-vectorized MIMD-concurrent workloads on NVRAM and compared their behavior to DRAM. Based on the revealed behavior we suggested recommendations for efficient deployment of available data parallelism primitives. Further, we proposed mixtures of various SIMD versions to be combined in a single query execution environment and showed that such cocktails could be leveraged via *adaptive SIMD-MIMD shaking*, our proposed approach allows to significantly increase the performance gains of vectorization for analytical workloads. We verified the effectiveness of our ideas through a prototypical implementation based on profiling and an inter-query concurrency mechanism.

ACKNOWLEDGMENTS

This work was partly funded by (1) the German Research Foundation (DFG) via an individual project LE-1416/27-1 as well as by a Reinhart Koselleck-Project (LE-1416/28-1), and (2) the European Union's Horizon 2020 research and innovation program under grant agreement No 957407 (DAPHNE).

REFERENCES

[1] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Found. Trends Databases* 5, 3 (2013), 197–280.

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *SIGMOD*. 671–682.

[3] Alfons Kemper Thomas Neumann Takushi Hashida Kazuichi Oe Yoshiyasu Doi Lilian Harada Sato Mitsuru Alexander van Renen, Viktor Leis. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*. 691–706.

[4] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.* 10, 12 (2017), 1754–1765.

[5] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016), 337–348.

[6] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-Juergen Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *OOPSLA*. 677–694.

[8] Carsten Binnig, Stefan Hildenbrand, and Franz Färber. 2009. Dictionary-based order-preserving string compression for main memory column stores. In *SIGMOD*. 283–296.

[9] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. 37–48.

[10] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.

[11] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB*. 54–65.

[12] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR*.

[13] Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. https://doi.org/10.1145/2933057.2933096

[14] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record* 26, 1 (1997), 65–74.

[15] Shimin Chen, Phillip B. Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *CIDR*. 21–31.

[16] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *PVLDB* 8, 7 (2015), 786–797.

[17] George P. Copeland and Setrag Khoshafian. 1985. A Decomposition Storage Model. In *SIGMOD*. 268–279.

[18] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46.

[19] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410.

[20] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *SOSP*. 478–493.

[21] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *EDBT*. 313–324.

[22] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *EuroSys*. 15:1–15:15.

[23] Georgios Giannikis, Darko Makreshanski, Gustavo Alonso, and Donald Kossmann. 2014. Shared Workload Optimization. *Proc. VLDB Endow.* 7, 6 (Feb. 2014), 429–440. https://doi.org/10.14778/2732279.2732280

[24] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. 2020. Automatic Core Specialization for AVX-512 Applications. In *SYSTOR*. 25–35.

[25] Dirk Habich, Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Juliana Hildebrandt, and Wolfgang Lehner. 2019. MorphStore - In-Memory Query Processing based on Morphing Compressed Intermediates LIVE. In *SIGMOD Conference*. 1917–1920.

[26] Intel. 2018. Intel Instruction Reference Manual (Vol 2A, 3-147). (2018).

[27] Intel. 2020. *Intel® Xeon® Processor Scalable Family, Specification Update*. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-scalable-spec-update.pdf

[28] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:cs.DC/1903.05714

[29] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *SOSP*. 494–508.

[30] Soroosh Khoram, Yue Zha, Jialiang Zhang, and Jing Li. 2017. Challenges and Opportunities: From Near-memory Computing to In-memory Computing. In *ISDP*. 43–46.

[31] Amandeep Khurana and Julien Le Dem. 2018. The Modern Data Architecture: The Deconstructed Database. *login Usenix Mag.* 43, 4 (2018).

[32] Wook-Hee Kim, Jihye Seo, Jinwoong Kim, and Beomseok Nam. 2018. clfB-tree: Cacheline Friendly Persistent B-tree for NVRAM. *ACM Trans. Storage* 14, 1 (Feb. 2018), 5:1–5:17.

[33] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.

[34] Thomas Kissinger, Tim Kiefer, Benjamin Schlegel, Dirk Habich, Daniel Molka, and Wolfgang Lehner. 2014. ERIS: A NUMA-Aware In-Memory Storage Engine for Analytical Workload. In *ADMS@VLDB*. 74–85.

[35] Rakesh Kumar, Alejandro Martínez, and Antonio González. 2014. Efficient Power Gating of SIMD Accelerators Through Dynamic Selective Devectorization in an HW/SW Codesigned Environment. *ACM Trans. Archit. Code Optim.* 11, 3 (2014), 25:1–25:23.

[36] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *SIGMOD*. 311–326.

[37] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Softw., Pract. Exper.* 45, 1 (2015), 1–29.

[38] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2016. MQJoin: Efficient Shared Execution of Main-Memory Joins. *Proc. VLDB Endow.* 9, 6 (Jan. 2016), 480–491. https://doi.org/10.14778/2904121.2904124

[39] Prashanth Menon, Andrew Pavlo, and Todd C. Mowry. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together At Last. *Proc. VLDB Endow.*

[40] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. 2013. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *TRIOS@SOSP*. 1:1–1:17.

[41] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *DaMoN*. 8:1–8:7.

[42] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. 371–386.

[43] Ismail Oukid and Wolfgang Lehner. 2017. Data Structure Engineering For Byte-Addressable Non-Volatile Memory. In *SIGMOD*. 1759–1764.

[44] Ismail Oukid and Wolfgang Lehner. 2017. Towards a Single-Level Database Architecture on Non-Volatile Memory. In *NVMW*.

[45] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *CIDR*.

[46] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. [n.d.]. Data-Oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1 ([n. d.]), 928–939.

[47] Kyriakos Paraskevas, Andrew Attwood, Mikel Luján, and John Goodacre. 2019. Scaling the Capacity of Memory Systems; Evolution and Key Approaches. In *MEMSYS*. 235–249.

[48] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules. In *MEMSYS*. 288–303.

[49] Ivy Bo Peng, Stefano Markidis, Erwin Laure, Gokcen Kestor, and Roberto Gioiosa. 2016. Exploring Application Performance on Emerging Hybrid-Memory Supercomputers. In *HPCC/SmartCity/DSS*. 473–480.

[50] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.

[51] Orestis Polychroniou and Kenneth A. Ross. 2014. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*. 755–766.

[52] Orestis Polychroniou and Kenneth A. Ross. 2020. VIP: A SIMD vectorized analytical query engine. *VLDB J.* 29, 6 (2020), 1243–1261.

[53] Iraklis Psaroudakis, Manos Athanassoulis, and Anastasia Ailamaki. 2013. Sharing Data and Work across Concurrent Analytical Queries. *Proc. VLDB Endow.* 6, 9 (July 2013), 637–648. https://doi.org/10.14778/2536360.2536364

[54] Lin Qiao, Vijayshankar Raman, Frederick Reiss, Peter J. Haas, and Guy M. Lohman. 2008. Main-memory scan sharing for multi-core CPUs. *Proc. VLDB Endow.* 1, 1 (2008), 610–621.

[55] Vijayshankar Raman, Gopi K. Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, René Müller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam J. Storm, and Liping Zhang. 2013. DB2 with BLU Acceleration: So Much More than Just a Column Store. *PVLDB* 6, 11 (2013), 1080–1091.

[56] Andy Rudoff. 2015. Persistent Memory Programming. *Login: The Usenix Magazine* 42 (2015), 34–40.

[57] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. In

[58] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *VLDB*. 553–564.

[59] 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 *CIDR*.

[60] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *FAST*. 5–5.

[61] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *PVLDB* 7, 5 (2014), 413–424.

[62] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *DBTest@SIGMOD*. 1:1–1:6.

[63] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads. In *BTW*. 313–333.

[64] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-Scan: Ultra Fast in-Memory Table Scan using on-Chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.

[65] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*. 323–338.

[66] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2019. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. arXiv:cs.DC/1908.03583

[67] Jun Yang, Qingsong Wei, Chundong Wang, Cheng Chen, Khai Leong Yong, and Bingsheng He. 2016. NV-Tree: A Consistent and Workload-Adaptive Tree Structure for Non-Volatile Memory. *IEEE Trans. Computers* 65, 7 (2016), 2169–2183.

[68] Songping Yu, Nong Xiao, Mingzhu Deng, Fang Liu, and Wei Chen. 2017. Redesign the Memory Allocator for Non-Volatile Main Memory. *J. Emerg. Technol. Comput. Syst.* 13, 3 (2017).

[69] Songping Yu, Nong Xiao, Mingzhu Deng, Yuxuan Xing, Fang Liu, Zhiping Cai, and Wei Chen. 2015. WAlloc: An efficient wear-aware allocator for non-volatile main memory. In *IPCCC*. 1–8.

[70] Mikhail Zarubin, Patrick Damme, Dirk Habich, and Wolfgang Lehner. 2020. Polymorphic Compressed Replication of Columnar Data in Scale-Up Hybrid Memory Systems. In *SYSTOR*. 98–110. https://doi.org/10.1145/3383669.3398283

[71] Mikhail Zarubin, Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. 2018. Efficient Compute Node-local Replication Mechanisms for NVRAM-centric Data Structures. In *DaMoN@SIGMOD* (Houston, Texas) *(DAMON '18)*. ACM, New York, NY, USA, Article 7, 9 pages. https://doi.org/10.1145/3211922.3211931

[72] Jingren Zhou and Kenneth A. Ross. 2002. Implementing database operations using SIMD instructions. In *SIGMOD*. 145–156.

[73] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *ICDE*. 59.