Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations

Georgios Psaropoulos
georgios.psaropoulos@epfl.ch
EPFL, Switzerland
SAP SE, Germany

Ismail Oukid
ismail.oukid@sap.com
SAP SE, Germany

Thomas Legler
thomas.legler@sap.com
SAP SE, Germany

Norman May
norman.may@sap.com
SAP SE, Germany

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL, Switzerland
RAW Labs SA, Switzerland

ABSTRACT

Non-Volatile Memory (NVM) technologies exhibit 4× the read access latency of conventional DRAM. When the working set does not fit in the processor cache, this latency gap between DRAM and NVM leads to more than 2× runtime increase for queries dominated by latency-bound operations such as index joins and tuple reconstruction. We explain how to easily hide NVM latency by *interleaving* the execution of parallel work in index joins and tuple reconstruction using *coroutines*. Our evaluation shows that interleaving applied to the non-trivial implementations of these two operations in a production-grade codebase accelerates end-to-end query runtimes on both NVM and DRAM by up to 1.7× and 2.6× respectively, thereby reducing the performance difference between DRAM and NVM by more than 60%.

CCS CONCEPTS

• **Information systems → Main memory engines**; *Point lookups*; *Query operators*; *Storage class memory*; • **Software and its engineering → Coroutines**; *Software architectures*; *Software performance.*

KEYWORDS

non-volatile memory, memory stalls, latency hiding, interleaved execution, coroutines

ACM Reference Format:
Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3329785.3329917

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329917

1 INTRODUCTION

Non-Volatile Memory (NVM) is a new class of byte-addressable memory technologies that offer the persistence and high density of storage media with sub-microsecond access latencies. These properties enable NVM to serve as main memory and storage at the same time, and have spurred a line of research investigating how to leverage them in database systems. Academic researchers have proposed systems, such as SOFORT [29], Peloton [10], and FOEDUS [22], that incorporate NVM both as memory and storage, whereas database vendors like SAP, Oracle, Microsoft, and Aerospike, have raced to announce NVM-enabled products [2, 3, 5, 7]. With the imminent availability of Intel Optane DC Persistent Memory Module (PMM) [4], i.e., Intel's NVM DIMM[1], the question is not *whether*, but *when* we will see a database running on NVM in production.

Preliminary works show how to leverage the properties of NVM with DRAM-like performance [7, 34]. Still, the 300 ns of NVM read latency (see Section 2 for details) pose a major performance challenge for latency-bound database operations, such as index joins and tuple reconstruction in the context of dictionary-encoded column stores [17, 20]. An index join consists of multiple pointer-chasing lookups in one index, while tuple reconstruction visits each column, retrieving the encoded row value and decoding it with a lookup to the corresponding dictionary. Both operations involve random memory accesses whose latency the processor cannot hide; this latency is only exacerbated by placing data on NVM, leading to significant slowdowns.

In this paper, we show how to use NVM in latency-bound operations at a fraction of the slowdown incurred due to the 300 ns latency. In the aforementioned cases of index joins and tuple reconstruction, the respective index and column lookups do not depend on each other, allowing the processor to "context switch" among lookups upon cache misses and continue executing instructions instead of waiting for data to be fetched from main memory. As prior work [21, 31, 32] on join-like operations demonstrates, this form of *interleaved execution* can be implemented in a practical manner using *coroutines*, i.e., functions that can suspend their execution and be resumed at a later point. Our work adds NVM latency to the performance equation, focusing on the following questions: (a) What is the performance difference between NVM and DRAM

[1] Unless otherwise noted, the term NVM in this paper refers to Intel Optane DC PMM.

for latency-bound operations? (b) How much does interleaved execution reduce this difference? We address both questions with the binary search microbenchmark of [31], as well as the end-to-end execution of two queries: a query with an `IN`-predicate and one with a simple `SELECT *` statement, representing index joins and tuple reconstruction respectively.

We make the following contributions:

- We introduce interleaved execution to tuple reconstruction. Contrary to previous works that interleaved lookups to a single data structure, tuple reconstruction is the first case, to the best of our knowledge, to show that we can interleave arbitrary codepaths accessing several data structures each.
- We show that NVM latency implies more than 2× slowdown for two performance-critical operations: index join and tuple reconstruction.
- We bridge the performance gap between NVM and DRAM:
  - For the binary search microbenchmark, interleaving yields 2.8×–5× speedups when the NVM-placed arrays are larger than the processor cache (38.5 MB), and exhausts NVM bandwidth using 18 physical cores.
  - For the two queries, interleaving improves runtime by up to 1.7× and 2.6× respectively on DRAM and NVM, reducing the gap between the two by more than 60%.

Our work corroborates the benefits of interleaved execution, demonstrating that NVM latency can be hidden given (a) independent work to execute, (b) sufficient memory-level parallelism supported by the processor, and (c) enough memory bandwidth.

2 NON-VOLATILE MEMORY IN DATABASES

Non-Volatile Memory (NVM), also called Storage-Class Memory or Persistent Memory, is a class of memory technologies that combine the low latency (although higher) and byte-addressability of DRAM with the non-volatility and large capacity of storage media. Examples of NVM include Phase-Change Memory (PCM) [26], Magnetic RAM (MRAM) [14], and Resistive RAM (RRAM) [15]. In the race to bring NVM to market, Intel recently announced the commercial availability of its Optane DC Memory technology in the DIMM form factor [4]: Optane DC Persistent Memory Modules (PMM) embed up to 512 GB NVM, i.e., double the capacity of the largest DRAM DIMMs available today (256 GB), enabling computer systems with more main memory per socket. However, Optane DC PMM (and NVM in general) exhibits higher access latency and lower bandwidth in comparison to DRAM, with writes being slower than reads. The latency and bandwidth of Optane DC PMM can be masked with the *Memory* mode of operation, which leverages DRAM as a cache and has no memory persistency; for persistency, applications can use the *App Direct* mode, which exposes the PMM as a storage device. The two modes are detailed in [18], while our work employs the *App Direct* mode.

Table 1 compares the characteristics[2] of DRAM, Optane DC PMM in the *App Direct* mode, and a state-of-the-art SSD. We measured the characteristics of DRAM and Optane DC PMM on a system with a second generation Intel Xeon Scalable processor (codenamed Cascade Lake, see Table 2) using the Intel Memory Latency Checker [1]. As the table indicates, while noticeably faster than SSDs, Optane DC PMM has 4× higher latency and 10× lower random read bandwidth compared to DRAM.

Table 1: Characteristics of DDR4 DRAM, Intel Optane DC PMM, and a 1 TB **Samsung 970 PRO SSD.**

	DRAM	Optane DC PMM	SSD
Read Latency	73 ns	300 ns	230 µs
Seq. Read BW	110 GB/s	36 GB/s	3.5 GB/s
Rand. Read BW	100 GB/s	10 GB/s	1.9 GB/s
Byte-addressable	Yes	Yes	No

To mitigate NVM's higher latency, researchers have investigated data structures and algorithmic trade-offs that we briefly discuss below. We limit our discussion to techniques that aim to hide the latency of NVM. Orthogonal optimizations that target efficient failure-atomicity handling are out of the scope of this paper.

To decrease the number of NVM writes, Chen et al. [12] proposed to replace the traditionally sorted B-Tree nodes with unsorted ones, thereby replacing the write-heavy sorted insertion with a simple append operation. On the downside, search operations have to scan the whole unsorted node, instead of a binary search in a sorted node. The authors further investigate partitioned hash joins for NVM and propose *virtual cache partitioning*, a technique that keeps compacted references to the records instead of physically copying them; this reduces the number of writes during partitioning, but increases the number of reads during the join phase because records are scattered. Moreover, Viglas [35] proposed a framework for trading writes for reads, applied to several sorting and join algorithms on NVM. The framework distinguishes between two classes of algorithms. In the first one, the input is split between a write-incurring and a write-limited part. The second class is based on lazy processing; these algorithms trade reads for writes, keeping track of the cost of reads versus savings ratio. When the cost exceeds the savings, the algorithm writes an intermediate result, and then starts a new lazy processing phase.

Building on their previous work, Chen et al. [13] proposed the wB-Tree, an NVM-based persistent B^+-Tree that keeps sorted indirection arrays to mitigate the fact that tree nodes are kept unsorted. Furthermore, Yang et al. [36] proposed the NV-Tree, a cache-conscious B-Tree that does not enforce the consistency of inner nodes since they can be rebuilt from the leaf nodes, thereby reducing the number of NVM writes. Later, Oukid et al. [30] presented the FPTree, a persistent B^+-Tree that keeps inner nodes in DRAM and leaf nodes in NVM, thereby avoiding NVM accesses while traversing inner nodes. The FPTree also keeps fingerprints in leaf nodes to minimize the number of cache lines that are touched in NVM. Other NVM-based structures that build on the above techniques and further optimize failure-atomicity handling have been proposed, e.g., the Write-Optimal Adaptive Radix Tree [27] and the Bz-Tree [8]. Following another approach, Van Renen [34] and Lersch et al. [28] investigated efficient buffering techniques for accessing NVM either through DRAM or directly. Additionally, Arulraj et al. [9] proposed

[2]We do not discuss write operations and endurance; this work focuses on read operations, whereas endurance is expected to be several years thanks to the *wear leveling* that Optane DC PMMs embed.

to adapt the cost model of query optimizers to take into account NVM's read/write asymmetry. Finally, Izraelevitz et al. [19] have recently studied the performance impact of Optane DC PMM on various systems. One of their observations is that workloads with random reads, which are the focus of our work, suffer significantly from Optane DC PMM's higher latency.

This paper complements these works, investigating a latency-hiding technique that can be combined with existing techniques.

3 INTERLEAVING WITH COROUTINES

In this section, we explain how to hide memory latency with interleaved execution. First, we revisit prior work that uses coroutines to implement interleaved execution, and then we introduce the extensions and optimizations necessary to effectively hide DRAM and NVM latency in arbitrary code.

3.1 Interleaved execution and coroutines

Interleaved execution is a universal latency-tolerance scheme [25] that leverages the presence of multiple independent tasks in data- and task-parallel code. This execution scheme avoids processor stalls in case of long-latency operations in one task, overlapping these operations with the execution of other tasks. To hide the latency of memory operations, software implementations of interleaved execution employ prefetch instructions to initiate asynchronous data fetches. Traditionally, interleaving has implied extensive code rewrites with techniques like *group prefetching* [11] and *asynchronous memory access chaining* [24], and has thus been avoided in production environments in favor of maintainability.

```
1  template<bool suspend>
2  task<int> binary_search(
3    vector<int>& array, int value
4  ) {
5    int low = 0; int size = array.size();
6    while(size > 1){
7      int probe = low + size/2;
8      int v = co_await
9        load<suspend>(array[probe]);
10     if(v < value){ low = probe; }
11     size -= size/2;
12   }
13   if(size == 1 && array[low] < value){ low++; }
14   if(array[low] == value) co_return low;
15   else co_return -1;
16 }
```

Listing 1: Binary search as a C++20 coroutine.

Recent proposals [21, 23, 31, 32] avoid the prohibitive code rewrites by encoding the independent tasks as coroutines, i.e., functions that *suspend* their execution at specified points and later *resume* from where they left off. Listing 1 hints the changes required to enable interleaved execution through an example depicting a binary search implemented as a C++20 coroutine [6]. The main change is at lines 8–9, where `array[probe]`, the array dereference that likely causes a cache miss for large `array` sizes, is replaced with `co_await load<suspend>(array[probe])`, an expression that, depending on the template parameter `suspend` [32], either loads `array[probe]` immediately, or issues a prefetch to `array[probe]`, suspends the execution of `binary_search` and loads the `array[probe]` value from the cache upon resumption. Given the lack of hardware support for informing memory operations [16], whether to fetch or to prefetch-and-suspend cannot be decided at runtime based on cache contents[3]; so the decision is static and relies on profiling to identify potential main memory accesses. Furthermore, the `return` keyword is changed to `co_return`, while the return type `task<int>` denotes a coroutine that holds an `int` result and keeps a pointer to its caller, enabling composition with other coroutines and propagation of suspensions/resumptions across call chains [21].

```
1  template<bool suspend>
2  vector<int> multiple_binary_searches(
3    int G,
4    vector<int>& array, vector<int>& values
5  ) {
6    vector<int> positions;
7    for_each<suspend>(G,
8      values.begin(), values.end(),
9      [&] (int value) -> root_task {
10       int position = co_await
11         binary_search<suspend>(array, value);
12       if(position != -1)
13         positions.push_back(low);
14   });
15   return output;
16 }
```

Listing 2: Interleaving binary searches.

Listing 2 illustrates the use of `binary_search` for multiple lookups in the same array. `multiple_binary_searches` has two parameters, the sorted `array` and the lookup `values`, and returns the `positions` of found values. The `for_each` is a variant of the *standard template library* (STL) algorithm that executes its last argument—here a lambda expression— for each `value` in `values` with or without interleaved execution, depending again on `suspend`. The lambda expression captures references to `positions` and `array` and returns a `root_task` coroutine, which is a simplified version of `task` that does not contain a result value nor maintains a pointer to its caller. The `co_await` expression in the lambda body has the following semantics: (a) `root_task` calls `binary_search`, which returns a `task<int>`; (b) `root_task` suspends, the pointer to caller of `task<int>` is set to the `root_task`, and `task<int>` resumes; (c) `root_task` is resumed by `task<int>` when the latter finishes execution. Note these suspensions are an essential part of converting call chain nodes from ordinary functions to coroutines, but have no direct connection to latency hiding. When the `task<int>` of `binary_search` suspends on a cache miss, execution control returns to `for_each`; `for_each` has a round-robin coroutine scheduler [31] managing a group of `G` coroutines running interleaved, where `G` is large enough to hide the memory latency [32]. After the evaluation of the `co_await` expression, the returned `position` is used to check if `array[position]` equals to `value`, in which case `position` is added to `positions`.

[3] With support for cache-content introspection, the suspension/resumption overhead could be avoided in case of a cache hit [31, 32]

The implementations presented in Listings 1 and 2 showcase the state of the art on how to use interleaving with coroutines to hide memory latency across instances of the same lookup task on one data structure. With the case of tuple reconstruction we discuss below, we introduce a set of extensions that facilitate interleaving for arbitrary codepaths.

3.2 Interleaving arbitrary codepaths

In the context of a column store like the one of SAP HANA, tuple reconstruction is a loop over the requested set of columns. Column stores often employ dictionary encoding, under which a column consists of a dictionary and a data vector, each having a different physical representation that depends on the stored datatype and the compression scheme used. Retrieving a value from such a column in a production-grade system involves a long call chain that differs from column to column, depending mainly on the data type and the compression scheme used in the dictionary and data vector implementations.

The long call chains in conjunction with the variety of codepaths substantially differentiate tuple reconstruction from index joins, which involve one data structure and one lookup implementation with few nested function calls. This differentiation precludes manual interleaving in the form of group prefetching [11] or asynchronous memory access chaining [24]: the first because static code rewriting cannot cover all column combinations for arbitrary schemas; and the second because suspensions at arbitrary depths in the call stack need to be surfaced to the function that iterates over the columns, requiring to convert all functions into state machines and thereby increase code complexity to extreme levels.

Still, where manual interleaving techniques fail, coroutines just work. We convert the functions involved in the column lookups into coroutines, in similar manner to the binary search example. However, we define `load` to accept an additional parameter, a lookup `context` that comprises the following:

- An optional `id` that identifies each column lookup and facilitates debugging.
- A low-overhead `allocator` for coroutine frames: given the size of all coroutine frames in a column lookup is bounded, we avoid the unnecessary overheads of a general purpose allocator by using a private-per-lookup, append-only allocator with preallocated memory. By reseting the allocator when the lookup finishes, we can reuse it in the next lookup.
- A reference to the coroutine scheduler of `for_each`. This reference enables `load` to directly resume the next lookup with a tail call.

Listing 3 depicts a simplified implementation of tuple reconstruction that interleaves the lookups to columns. To reconstruct the tuple that corresponds to a given `key`, we first look for the matching `row` in the `KEY` column of table `TBL` (line 2). Then we iterate over all columns (lines 4–10), and in each column we look for the corresponding value by calling `get` with the provided `context` (which is distinct per iteration) and the `row` as arguments (line 9); we store the retrieved value in the appropriate `tuple` position (`tuple[col.id]`). We should note here that `context` simplifies the implementation of `root_task`, which was originally [21] required to keep track of the current suspended leaf coroutine.

```
1   tuple_t reconstruct(int key) {
2     int row = TBL.columns[KEY].find(key);
3     tuple_t tuple;
4     for_each(G,
5       TBL.columns.begin(), TBL.columns.end(),
6       [&] (context_t& ctx, column_t& col)
7         -> root_task {
8         tuple[col.id] =
9           co_await col.get(ctx, row);
10    });
11    return tuple;
12  }
13
14  task<value_t> column_t::get(
15    context_t& ctx, int id
16  ) {
17    int code = co_await load(ctx, codes[id]);
18    co_return co_await dict.decode(ctx, code);
19  }
```

Listing 3: Interleaving tuple reconstruction.

We already mentioned that each column implementation is different depending on the datatype and the compression scheme of the column. In Listing 3, we also present an implementation of the `column_t::get` method for a dictionary-encoded column: we first `load` the encoded row value from a `codes` array (line 17) and then `decode` it using the dictionary `dict` (line 18)—the array access causes one cache miss, so we use the `load` to prefetch, suspend, and load, whereas `decode` causes one or more cache misses depending on the `dict` implementation. Despite its simplicity, this example is representative of the code changes necessary also for production-level column implementations, such as the ones of SAP HANA—the only difference is the number of functions we need to convert into `task`s.

```
1  task<int> f_coroutine(context_t& context) {
2    co_return transform(co_await g(context));
3  }
4
5  wrapped<int> f_function(context_t& context) {
6    return wrapped<int>{g(context), transform};
7  }
```

Listing 4: Example of coroutine elision.

Coroutine elision. A column lookup comprises many small functions that are inlined by the compiler. Essential to good performance is a compiler that inlines also the `task` counterparts of these functions, eliding a plethora of small coroutine allocations [33] and the associated instructions that manage coroutine lifetime.

At the time of writing, no compiler can reliably inline tasks, so we systematically replace each task that **`co_await`**s one nested task, with an ordinary function that has no **`co_await`** nor **`co_return`** in its body. Consider the example in Listing 4: `f_coroutine` passes the result of the **`co_await`** expression to a `transform` function before returning it (line 2). We convert `f_coroutine` to `f_function` by wrapping the `task<int>` returned from `g` along with the function `transform` in a `wrapped<int>` object. This object is not a separate

coroutine, but a wrapper that can participate in a **co_await** expression, with the distinctive property of transforming the result of the wrapped coroutine before returning it. One variation of this pattern has no result transformation, in which case the nested coroutine can be immediately returned without a wrapper. In other cases, there are two or more code branches that **co_await** different nested coroutines and apply distinct result transformations each; to unify the code branches, we define the wrapper to accept not only different transformations but also coroutine types other than `task`. Furthermore, to ensure a task does not outlive its parameters—a likely case when we elide coroutines—we move call parameters either to the nested coroutine or the wrapper. These considerations increase implementation complexity, but, as compiler support for coroutines matures, we expect coroutine elision to become a job for the compiler and not the programmer.

3.3 Interleaving accesses to NVM

Everything described so far about interleaving with coroutines applies to any use case with data- or task-level parallelism in which memory latency is exposed to runtime. What changes by placing data and/or working set to NVM, is the 4× latency increase and the 10× bandwidth reduction (see Section 2). To hide the higher latency, more instructions are needed. These instructions can be found by increasing the group size, i.e., the number of interleaved coroutines. However, the memory-level parallelism (MLP) of current Intel processors is 10 in-flight memory requests per core [18]. This limit means higher group sizes offer little to no benefit in case all prefetches go to main memory. Furthermore, scaling interleaved execution to multiple cores is bound to reach the bandwidth limit of NVM—even in the absence of scans. Under these constrains, interleaving converts execution from latency- to MLP- or bandwidth-bound, as we demonstrate next.

4 EVALUATION

In this section, we show that interleaving with coroutines drastically narrows the performance gap between NVM and DRAM for latency-bound operations despite the significant latency difference. To that end, we interleave two types of operations: (a) lookups on a single index, and (b) lookups on different indexes. In Section 4.1, we compare the single-thread runtime and the scalability of interleaved and non-interleaved binary searches having sorted arrays on DRAM and NVM. Then, we assess the gap between DRAM and NVM and the effect of interleaving on the end-to-end execution of two queries running on a prototype based on SAP HANA: a query with an `IN`-predicate resembling an index join (Section 4.2), and a simple SELECT(*) query (Section 4.3), which is the poster child of tuple reconstruction.

We compile the microbenchmarks and the prototype with Clang 7.0.1 using the `-fcoroutines-ts` option that enables coroutine support, our version of the `experimental/coroutine` header from the C++ standard library of LLVM (*libcxx*), which works with the widely available GNU C++ Library (*libstdc++*), and the `task` type from Lewis Baker's *cppcoro* library[4]. We run our experiments on a dual socket system equipped with an Intel Xeon Platinum 8280L processor, 6 DRAM DIMMs, and 6 Optane DC PMMs per socket (see Table 2) and SUSE Linux Enterprise Server 15 (Linux kernel 4.12); we dedicate one socket to our experiments, pinning all other processes to the other socket.

[4]https://github.com/lewissbaker/cppcoro

Processor	**Intel Xeon Platinum 8280L** (codename Cascade Lake)
Architecture	Cascade Lake
Technology	14 nm @ 2.7 GHz (up to 4 GHz)
# Cores (per socket)	28
L1 I/D (per core)	32 kB/32 kB, 8-way associative
# Line Fill Buffers	10
L2 (per core)	1 MB, 16-way associative
L3 (per core)	38.5 MB, 11-way associative, non-inclusive, victim cache
DTLB (4 kB/2 MB/1 GB pages)	64/32/4 entries, 4-way/4-way/fully associative
STLB (4 kB/2 MB/1 GB pages)	1536/1536/16 entries, 12-/12-/4-way associative
DRAM (per socket)	96 GB (6 × 16 GB)
Optane DC PMM (per socket)	768 GB (6 × 128 GB)

Table 2: Architectural parameters.

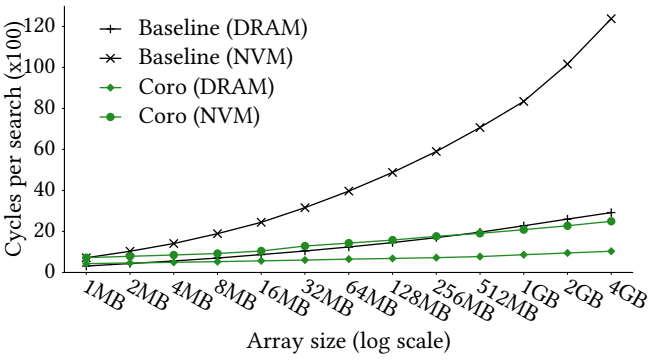

Figure 1: Binary search performance on DRAM vs NVM.

4.1 Microbenchmarks

We use the binary search microbenchmark described in [31], as representative of a simple index join. We use two implementations that look for a list of values in a sorted array of 32-bit signed (`int32_t`) integers. The first implementation, `Baseline`, executes one binary search after the other for each of the list values, while the second implementation, `Coro`, interleaves the execution of `G` binary searches at a time using coroutines, as described in [31]. The sorted array sizes range between 1 MB–4 GB, increased by 1 kB to circumvent alignment issues that hurt TLB performance [31]. For the lookup list we select 10'000 values from the sorted array using `std::mt19937` with a fixed seed of value 0 and `std::uniform_int_distribution`. We run our experiments with the array placed first in DRAM and then in NVM.

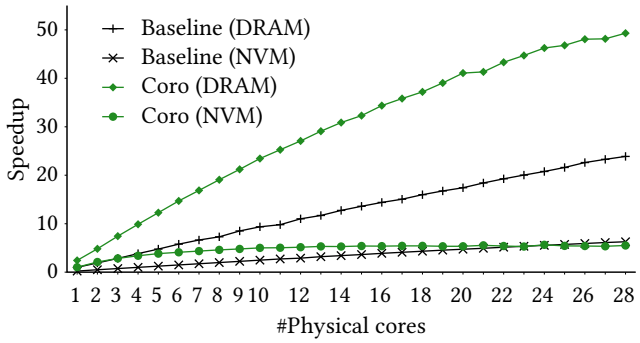

Figure 2: Scalability of binary search for a 2 GB sorted array.

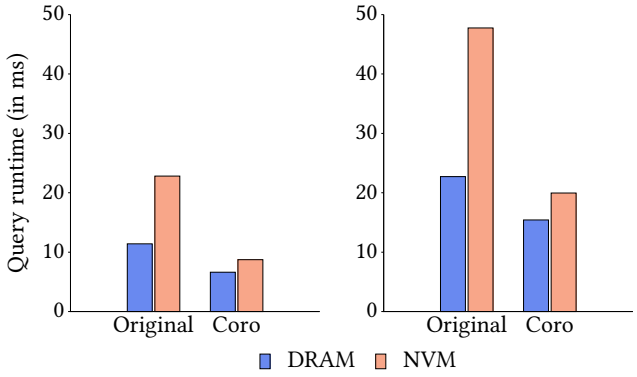

Figure 3: `IN`-predicate query on tables with 100M rows.

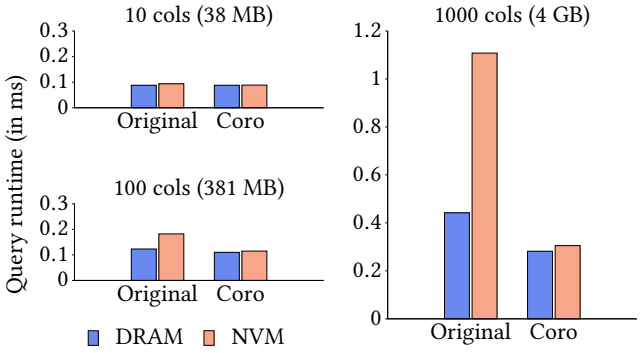

Figure 4: 'SELECT (*)' query on `INTEGER` tables with 1M rows and varying column counts.

Increasing the array size. Figure 1 depicts the cycles per binary search for all array sizes and for the optimal group sizes, i.e., the one for which `Coro` has the best performance. First, for `Baseline`, we notice the performance gap between `DRAM` and `NVM`, a gap that widens from 2.3× to 4.2× as the number of cache misses increases with the array size. For `Coro`, the performance difference ranges from 1.7× at 1 MB to 2.4× at 4 GB. The corresponding optimal group sizes are in the range 21–44 for DRAM and 22–49 for NVM. We observe values in the upper part of these ranges with small arrays and, conversely, values in the lower part with large arrays. For small arrays most accesses hit in the cache, allowing to interleave more than 10 coroutines at a time and thereby hide most of the latency of the few main memory accesses. As the number of main memory accesses increases, the hardware-imposed limit on in-flight memory requests becomes a bottleneck and decreases the optimal group size; still, the group size is above 20 because array values of the first binary search iterations fit in the cache, so the respective prefetches finish fast, allowing new prefetches to be issued[5]. The hardware-imposed limit explains also why interleaved execution does not eliminate the difference: the 7 assembly instructions that exist between subsequent array lookups in a binary search are inadequate for eliminating the latency gap given the group size limit. Still, interleaving improves NVM performance by up to 5× for 4 GB arrays, reaching runtimes similar to `Baseline` on DRAM.

Scaling up to 28 cores. We assess the scalability of inteleaved execution on NVM by running a multithreaded version of the above microbenchmark on one socket using 1–28 physical cores. Figure 2 shows the speedups over `Baseline (DRAM)` with 1 core. Contrary to the latency-bound `Baseline`, that scales well on both DRAM and NVM, `Coro` on NVM becomes bandwidth-bound with 18 cores, reaching a maximum speedup of 8× and incurs a slight slowdown as the number of cores used increases further to 28 due to increasing resource contention on the socket; on DRAM performance scales sublinearly up to 50×.

[5]Note that the difference to the optimal group size 10 reported in [31] is due to the higher suspension/resumption overhead of the code generated by the Microsoft Visual C++ compiler. Compiler support matters and is improving over the years.

4.2 Index join

In addition to the binary search microbenchmark, we evaluate the respective effects of NVM latency and interleaved execution on the semijoin of `IN`-predicate clauses involving dictionary-encoded columns [31]. We run the following `IN`-predicate query on our prototype: `SELECT COUNT(*) FROM TBL WHERE COL IN LIST`, where `TBL` has 100M rows and is placed first in DRAM and then in NVM, `LIST` contains 10000 randomly generated values, and for `COL` we analyze two datatypes: either `INTEGER` or `VARCHAR(15)`. In Figure 3, we report the average runtimes of 1000 executions per datatype. For the `Original` implementations and both datatypes, NVM runtime is 2× the DRAM runtime, while coroutines (`Coro`) reduce the difference to 30%—the difference is not eliminated because the index lookup is a variant of binary search with additional compression-related indirection and thus lacks the amount of work required to completely hide NVM latency. Still, `Coro` performs better than `Original`.

4.3 Tuple reconstruction

To evaluate the effect of interleaved execution on tuple reconstruction, we use a `SELECT(*) FROM TBL WHERE KEY=X` query, where `KEY` has unique values.

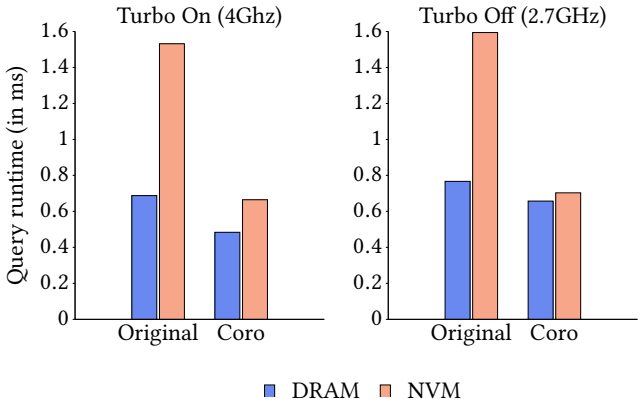

Figure 5: 'SELECT (*)' query on a table with 1M rows and 1000 columns (of INTEGER, DECIMAL(10,2), and NVARCHAR(50) type), with and without frequency scaling.

Ranging column count. We assess the worst-case slowdown due to NVM by executing the query on a `TBL` with 1M rows and 10, 100, and 1000 `INTEGER` columns with unique values each. Retrieving the value from each `INTEGER` column incurs two cache misses—one when accessing the data vector and another one when accessing the dictionary—that dominate execution. In Figure 4, we depict the average runtime of 10000 query executions, having placed `TBL` on DRAM and NVM, with (`Coro`) and without (`Original`) interleaving. We observe the runtime gap between `Original` and `Coro` widen from 6% to 150%, as the work in the reconstruction loop increases along with the column count; for `Coro`, the gap is 1%–8% and not eliminated due to lack of instructions. More interestingly, for 1000 columns, `Coro` on NVM is 30% faster than `Original` on DRAM.

Mixed datatypes. We show interleaving of three distinct codepaths by executing the query on a table `TBL` of 1000 columns, of which one third have datatype `INTEGER`, another third `DECIMAL(10,2)`, and the last third `VARCHAR(50)`. Moreover, we highlight the effect of processor frequency on execution by running the experiment with frequency scaling enabled (`Turbo On`) and disabled (`Turbo Off`). In Figure 5, we see interleaved execution reduces the performance gap from 123% to 37% for `Turbo On`. Lookups in `DECIMAL` and `VARCHAR` columns involve more instructions compared to `INTEGER` columns, explaining the higher runtime compared to the 1000 column case in Figure 4. However, the dictionaries of `VARCHAR` columns use prefix compression, which means dictionary lookups first retrieve the prefix and then the rest of the value with an additional indirection; contrary to the latter access that is a guaranteed cache miss for different values, the prefix is found in cache often enough to not justify a blind suspension for DRAM accesses, leading to the 37% gap for `Coro`—hardware support for cache content introspection would be beneficial for this case. Again, `Coro` runtime on `NVM` is faster than `Original` on `DRAM`. Finally, for `Turbo Off`, we see the gap reduces from 106% to 7% due to the lower frequency: the work needed to hide a given latency is less at 2.7 GHz than at 4 GHz.

5 CONCLUSIONS

Non-volatile memory brings higher main memory capacities with a latency 4× higher than DRAM. In this paper, we have showed how to bridge this latency gap and facilitate the adoption of NVM for latency-bound workloads by leveraging the abundant parallel work present in these workloads. We hide most of the NVM latency by interleaving the execution of this work in a practical manner, using language support in the form of coroutines. In the end, NVM latency is just a higher latency that can be hidden given enough parallel work and hardware resources.

ACKNOWLEDGMENTS

We would like to thank our Intel colleagues, Thomas Willhalm and Roman Dementiev, for providing access to Cascade Lake systems equipped with Intel Optane DC Persistent Memory, and for their key role in understanding the behavior of these systems. We also thank the anonymous reviewers, as well as Angelos Anadiotis, Periklis Chrysogelos, Stella Giannakopoulou, and Stefan Noll for their valuable feedback.

REFERENCES

[1] 2013. Intel Memory Latency Checker. http://www.intel.com/software/mlc [Online; accessed 18-March-2019].
[2] 2019. Aerospike 4.5: Persistent Memory and Compression. https://www.aerospike.com/blog/aerospike-4-5-persistent-memory-compression/ [Online; accessed 18-March-2019].
[3] 2019. How to configure persistent memory (PMEM) for SQL Server on Linux. https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-configure-pmem?view=sqlallproducts-allversions [Online; accessed 18-March-2019].
[4] 2019. Intel Optane DC Persistent Memory Module. www.intel.com/optanedcpersistentmemory [Online; accessed 20-March-2019].
[5] 2019. Oracle TimesTen In-Memory Database. https://www.oracle.com/database/technologies/related/timesten.html [Online; accessed 18-March-2019].
[6] 2019. Working Draft, Standard for Programming Language C++. http://eel.is/c++draft/dcl.fct.def.coroutine [Online; accessed 25-March-2019].
[7] 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 (Aug. 2017), 1754–1765. https://doi.org/10.14778/3137765.3137780
[8] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proceedings of the VLDB Endowment* 11, 5 (2018), 553–565.
[9] Joy Arulraj and Andrew Pavlo. 2017. How to Build a Non-Volatile Memory Database Management System. In *Proceedings of the 2017 ACM International Conference on Management of Data*. ACM, 1753–1758.
[10] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *PVLDB* 10, 4 (2016), 337–348.
[11] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving Hash Join Performance Through Prefetching. *ACM Trans. Database Syst.* 32, 3 (2007).
[12] Shimin Chen, Phillip B Gibbons, and Suman Nath. 2011. Rethinking Database Algorithms for Phase Change Memory. In *Fifth Biennial Conference on Innovative Data Systems Research (CIDR)*. 21–31.
[13] Shimin Chen and Qin Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.
[14] Xiangyu Dong, Xiaoxia Wu, Guangyu Sun, Yuan Xie, Helen Li, and Yiran Chen. 2008. Circuit and microarchitecture evaluation of 3D stacking magnetic RAM (MRAM) as a universal memory replacement. In *45th ACM/IEEE Design Automation Conference*. IEEE, 554–559.
[15] B Govoreanu, GS Kar, YY Chen, V Paraschiv, S Kubicek, A Fantini, IP Radu, L Goux, S Clima, R Degraeve, et al. 2011. 10× 10nm 2 Hf/HfO x crossbar resistive RAM with excellent performance, reliability and low-energy operation. In *IEEE International Electron Devices Meeting (IEDM)*. IEEE, 31–6.
[16] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. 1996. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. *SIGARCH Comput. Archit. News* 24, 2 (May 1996), 260–270. https://doi.org/10.1145/232974.233000

[17] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. 2009. Self-organizing Tuple Reconstruction in Column-stores. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 297–308. https://doi.org/10.1145/1559845.1559878

[18] Intel Corporation. 2019. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*.

[19] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[20] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. *Proceedings - International Conference on Data Engineering*, 466–475. https://doi.org/10.1109/ICDE.2007.367892

[21] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *PVLDB* 11, 11 (July 2018), 1702–1714. https://doi.org/10.14778/3236187.3236216

[22] Hideaki Kimura. 2015. FOEDUS: OLTP engine for a thousand cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 691–706.

[23] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. 2018. Cimple: Instruction and Memory Level Parallelism: A DSL for Uncovering ILP and MLP. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, Article 30, 16 pages. https://doi.org/10.1145/3243176.3243185

[24] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *PVLDB* 9, 4 (2015), 252–263. https://doi.org/10.14778/2856318.2856321

[25] James Laudon, Anoop Gupta, and Mark Horowitz. 1994. Interleaving: A Multithreading Technique Targeting Multiprocessors and Workstations. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*. ACM, New York, NY, USA, 308–318. https://doi.org/10.1145/195473.195576

[26] Benjamin C Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (2010).

[27] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. 2017. WORT: Write Optimal Radix Tree for Persistent Memory Storage Systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. 257–270.

[28] Lucas Lersch, Ismail Oukid, Ivan Schreter, and Wolfgang Lehner. 2017. Rethinking DRAM caching for LSMs in an NVRAM environment. In *European Conference on Advances in Databases and Information Systems*. Springer, 326–340.

[29] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A hybrid SCM-DRAM storage engine for fast data recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*. ACM, 8.

[30] 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 *Proceedings of the 2016 ACM International Conference on Management of Data (SIGMOD)*. ACM, 371–386.

[31] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB* 11, 2 (Oct. 2017), 230–242. https://doi.org/10.14778/3149193.3149202

[32] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2018. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal* (14 Dec 2018). https://doi.org/10.1007/s00778-018-0533-6

[33] Richard Smith and Gor Nishanov. 2018. Halo: coroutine Heap Allocation eLision Optimization: the joint response. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0981r0.html [Online; accessed 15-March-2019].

[34] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 1541–1555.

[35] Stratis D Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proceedings of the VLDB Endowment* 7, 5 (2014), 413–424.

[36] J. Yang, Q. Wei, C. Wang, C. Chen, K. Yong, and B. He. 2015. NV-Tree: A Consistent and Workload-adaptive Tree Structure for Non-volatile Memory. *IEEE Trans. Comput.* PP, 99 (2015).