Micro-architectural Analysis of OLAP: Limitations and Opportunities

Utku Sirin
EPFL
utku.sirin@epfl.ch

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

ABSTRACT

Understanding micro-architectural behavior is important for efficiently using hardware resources. Recent work has shown that in-memory online transaction processing (OLTP) systems severely underutilize their core micro-architecture resources [29]. Whereas, online analytical processing (OLAP) workloads exhibit a completely different computing pattern. OLAP workloads are read-only, bandwidth-intensive, and include various data access patterns. With the rise of columnstores, they run on high-performance engines that are tightly optimized for modern hardware. Consequently, micro-architectural behavior of modern OLAP systems remains unclear.

This work presents a micro-architectural analysis of a set of OLAP systems. The results show that traditional commercial OLAP systems suffer from their long instruction footprint, which results in high response times. High-performance columnstores execute tight instruction streams; however, they spend 25 to 82% of their CPU cycles on stalls both for sequential- and random-access-heavy workloads. Concurrent query execution can improve the utilization, but it creates interference in the shared resources, which results in sub-optimal performance.

PVLDB Reference Format:
Utku Sirin and Anastasia Ailamaki. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *PVLDB*, 13(6): 840-853, 2020.
DOI: https://doi.org/10.14778/3380750.3380755

1. INTRODUCTION

Online analytical processing (OLAP) is an ever-growing, multi-billion dollar industry. To extract valuable information from their data, many industrial and community organizations rely on fast and efficient analytical processing. Micro-architectural behavior reveals the limitations of and opportunities for efficiently using modern hardware resources, hence enables the delivery of high performance. Research has shown that OLAP systems can improve performance by orders of magnitude by more efficiently using the modern hardware resources [23].

Micro-architectural behavior of online transaction processing (OLTP) workloads has been studied extensively. Recent work has shown that, despite being aggressively optimized for modern hardware, in-memory OLTP systems spend the majority of their time in instruction and/or data cache misses [29]. Whereas, OLAP workloads exhibit a completely different computing pattern. Unlike the update-heavy OLTP workloads, OLAP workloads are read-only. Therefore, they do not require a concurrency control and logging mechanism or a complex buffer pool for synchronizing the modified pages on disk. OLAP workloads are arithmetic-operation- and bandwidth-intensive. They process large amounts of data with various data access patterns including both sequential and random data accesses.

With the rise of columnstores [1, 9], researchers proposed a diverse set of query processing paradigms (vectorized [5, 24] vs. compiled query processing [17]), and system prototypes (Proteus [16], Typer, and Tectorwise [18]). Many database systems, such as SQL Server, Oracle, and DB2, support a columnstore extension [19, 20, 28]. Columnstores operate only on the columns that are necessary for the query, thus utilize memory bandwidth more efficiently. They process columns in tight, hardware-friendly execution loops that are optimized for the efficient use of the CPU cycles.

The micro-architectural behavior of modern OLAP systems is unclear. In this paper, we perform a detailed micro-architectural analysis of OLAP workloads running on modern hardware. We profile six systems that vary from a traditional commercial row-store to high-performance execution engines. We evaluate the breakdown of the CPU cycles, memory bandwidth utilization, and normalized end-to-end response time. We examine how well each OLAP system uses the hardware resources and, in terms of delivering high performance, what the limitations and opportunities are. In this paper, we show the following:

- Unlike the traditional commercial OLTP systems, traditional commercial OLAP systems and their columnstore extensions do not suffer from instruction cache misses. Nevertheless, they carry high instruction overheads and deliver performance orders of magnitude lower than state-of-the-art columnstores.

- State-of-the-art columnstores use numerous optimization techniques such as SIMD, predication and bloom filters. However, they spend 25 to 82% of their CPU cycles to stalls. Scan-intensive queries suffer from ba-

Proceedings of the VLDB Endowment, Vol. 13, No. 6
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3380750.3380755

ndwidth-bounded data cache stalls, while join-intensive queries suffer from latency-bounded data cache stalls.

- Scan-intensive queries on the state-of-the-art column-stores saturate the memory bandwidth before saturating the cores. Join-intensive queries saturate the cores before saturating the memory bandwidth. Concurrently executing scan- and join-intensive queries enables the saturation of both the cores and bandwidth. However, this creates interference in the shared memory bandwidth, hence results in sub-optimal performance.

The paper is organized as follows. In Section 2, we present the experimental setup and methodology. In Section 3, 4 and 5, we present the projection, selection and join microbenchmark analyses. In Section 6, we present the analysis of TPC-H queries. In Section 7, we present mixed query workload analysis. In Section 8, 9, 10 and 11, we present the analyses of predication, SIMD, hardware prefetchers and hyper-threading/turbo-boost. In Section 12, we summarize the lessons learned. Lastly, In Section 13 and 14, we present the related work and conclusion.

2. SETUP & METHODOLOGY

In this section, we present the experimental setup and methodology.

Benchmarks: We use microbenchmarks and TPC-H queries [35]. We use projection, selection, and join microbenchmarks as they constitute the basic SQL operators. All the systems use the hash join algorithm when running the join microbenchmark.

All the microbenchmarks use the TPC-H schema. The projection microbenchmark does a single SUM() over a set of columns from the lineitem table. We vary the number of columns from one to four. We use *l_extendedprice*, *l_discount*, *l_tax* and *l_quantity* columns. We add the projected columns inside the SUM(). We call the projection microbenchmark that does a SUM() over n columns a projection query with the degree of n.

The selection microbenchmark extends the projection query with the degree of four with a WHERE clause of three predicates over three columns of the lineitem table: *l_shipdate*, *l_commitdate* and *l_receiptdate*. It varies the selectivity of each individual predicate from 10% to 50% and 90%. The join microbenchmark does a join over two tables, followed by a projection. The small-sized join microbenchmark joins the supplier and nation tables, it and does a SUM() over the addition of *s_acctbal* and *s_suppkey*. The medium-sized join joins the partsupplier and supplier tables, and it does a SUM() over the addition of *ps_availqty* and *ps_supplycost*. The large-sized join joins the lineitem and orders table, and it does a SUM() over the addition of the four columns that the projection query with the degree of four uses.

We profile a large subset of TPC-H queries on DBMS V. We chose DBMS V for this purpose, as DBMS V is the highest performing real-life system we use. We categorize the TPC-H queries based on their micro-architectural behavior. We then choose six representative queries and continue with the cross-system analysis. Our selection of the queries corroborates with the queries used by [18].

Hardware: We conduct our experiments on an Intel Broadwell server. Table 1 presents the server parameters. As the Broadwell micro-architecture does not support AVX-512 instructions, we conduct the SIMD experiments on a separate Skylake server. The Skylake server has a similar execution engine but a different memory hierarchy from the Broadwell server. The Skylake server has a significantly larger L2 cache (1 MB), a smaller non-inclusive L3 cache (16MB), a smaller per-core (10 GB/s) and a larger per-socket (87 GB/s) sequential access bandwidth. It has a similar per-core and per-socket random access bandwidth.

Table 1: Broadwell server parameters.

Processor	Intel(R) Xeon(R) CPU E5-2680 v4 (Broadwell)
#sockets	2
#cores per socket	14
Hyper-threading	Off
Turbo-boost	Off
Clock speed	2.40GHz
Per-core bandwidth	12GB/s (sequential) 7GB/s (random)
Per-socket bandwidth	66GB/s (sequential) 60GB/s (random)
L1I / L1D (per core)	32KB / 32KB 16-cycle miss latency
L2 (per core)	256KB 26-cycle miss latency
L3 (shared)	(inclusive) 35MB 160-cycle miss latency
Memory	256GB

We use Intel's Memory Latency Checker (MLC) [11] to measure cache access-latencies and maximum single/multi-core and random/sequential-access bandwidth.

OLAP systems: We examine (i) a commercial row-store, DBMS R, (ii) the columnstore extension of the commercial row-store, DBMS C, (iii) an open-source, full-fledged OLAP system, Quickstep [24], (iv) a popular, high-performance, commercial columnstore based on vectorized query processing, DBMS V, (v) an open-source OLAP engine based on vectorized query processing, Tectorwise [18], and (vi) an open-source OLAP engine based on data-centric code generation, i.e., compiled, query processing, Typer [18]. We chose these six systems as each represents a different category of a system and execution model.

OS & Compiler: We use Ubuntu 16.04.6 LTS and gcc 5.4.0 on the Broadwell server, and Ubuntu 18.04.2 LTS and gcc 7.4.0 on the Skylake server.

VTune: We use Intel VTune 2018 on the Broadwell server, and VTune 2019 on the Skylake server. We use VTune's built-in general-exploration (uarch-exploration on VTune 2019) analysis for the breakdown of the CPU cycles. We use VTune's built-in memory-access analysis to measure the consumed memory bandwidth. As we numa-localize our experiments on a single socket, we report average bandwidth per-socket values. We use VTune's built-in hotspots and advanced-hotspots analyses to perform function call trace breakdown.

Background: VTune's general-exploration provides the breakdown of the CPU cycles [30, 36]. Each CPU cycle can be categorized into one of two classes: *retiring* and *stalling*. A retiring cycle is a cycle where the processor finishes the execution of an instruction, i.e., *retires an instruction*. A

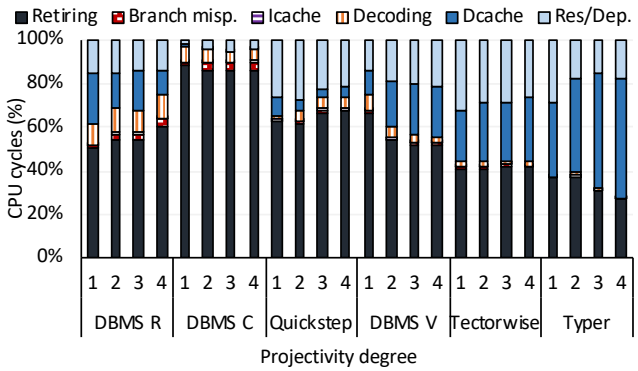

Figure 1: Breakdowns of the CPU cycles for projection microbenchmark for single-threaded execution.

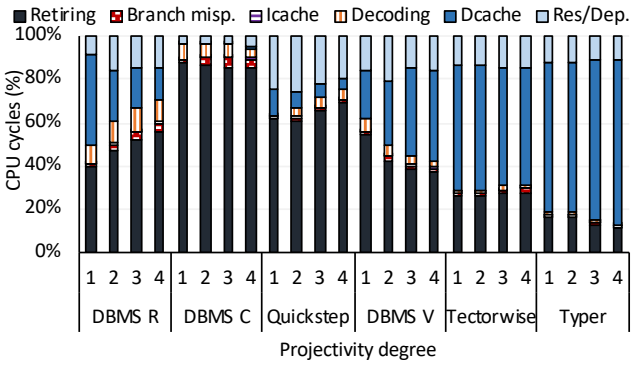

Figure 2: Breakdowns of the CPU cycles for projection microbenchmark for multi-threaded execution.

stalling cycle is a cycle where the processor has to wait, i.e., *stall*, (e.g., to perform a read from the caches). In an ideal scenario, all CPU cycles would be retiring.

Stalling cycles can be further decomposed into five components: (i) branch mispredictions, (ii) Icache, (iii) decoding, (iv) Dcache and (v) resource/dependency. Branch mispredictions define the cost for mispredicted branch instructions. Today's processors use a hardware unit called branch predictor; it predicts the outcome of a branch instruction (i.e., an `if()` statement) and continues executing the instructions as if the branch was correctly predicted. If the processor then realizes the prediction is not correct, it undoes whatever it has been doing and starts executing the correct set of instructions. This cost is defined as the branch mispredictions and can be very costly, as it requires canceling a large amount of work. Icache defines the cost of instruction cache misses. Decoding defines the cost of sub-optimal micro-architectural implementation of the instruction decoding unit. Dcache defines the cost of data-cache misses. Resource/dependency defines the cost of executing instruction that has resource and/or data dependencies.

Measurements: For every experiment, we first populate the database. We use a one-minute warmup period, followed by a three-minute VTune profiling period. We disable hyper-threading (HT) and turbo-boost (TB), as they jeopardize VTune counter values [12]. We examine HT and TB separately, in Section 11.

We numa-localize every experiment by using Linux's numactl command. We do single- and multi-threaded experiments. For the multi-threaded experiments, we use the number of threads that provides the lowest response time. We choose a scaling factor of 70 (the database of 70GB) for all the experiments as it makes 5GB/core to process; this is large enough for out-of-cache experiments.

We generate statistics before profiling each database. For a more fair comparison, we disable compression for all the systems. We test the compression on DBMS V when it runs the TPC-H benchmark, and we see that it increases the response time for 18 of the 22 queries. For the remaining 4 queries, it decreases the response time less than 15%.

We do hardware prefetcher experiments in Section 10 by modifying the relevant model-specific register (msr) of the processor [10].

3. PROJECTION

We present the projection microbenchmark. Figure 1 and 2 show the breakdowns of the CPU cycles, Table 2 presents normalized response times, and Table 3 presents consumed memory bandwidth values for single- and multi-threaded executions.

DBMS R & C spend the majority of the CPU cycles retiring instructions without any Icache stalls. This shows that, unlike commercial OLTP systems that suffer mostly from Icache stalls; commercial OLAP systems do not suffer from Icache stalls [29]. Table 2 shows, however, that DBMS R and C are 10 to 56 times slower than the state-of-the-art columnstore DBMS V. As the number of retiring cycles is proportional to the number of retired instructions, DBMS R and C, nevertheless, suffer mainly from their large instruction footprints.

Quickstep spends the majority of the CPU cycles retiring instructions. Quickstep is 1.7 to 2.8 times slower than DBMS V. This shows that Quickstep carries instruction overhead. We examine Quickstep's function-call trace for the projection query of degree four. Quickstep spends 50% of its time in `getUntypedValue()` and 8.1% of its time in `next()` function. It spends the remaining time inside a functor that performs aggregation. `getUntypedValue()` function performs null/boundary checking, whereas `next()` is used to increment the processed tuple ID.

Quickstep relies on aggressive function inlining, where at every iteration of the aggregation it makes inlined function calls of `getUntypedValue()` and `next()` per tuple. Although inlined function calls are not as expensive as regular or virtual function calls, they require extra work. `getUntypedValue()` makes five further inlined function calls, which requires even more work, hence making the aggregation 1.7 to 2.8 times slower than DBMS V.

DBMS V is twice as slow as Tectorwise at the projectivity of degree one for single-threaded execution. As DBMS V and Tectorwise implement a similar execution model, the difference highlights the overhead that a full-fledged, real-life system should pay off. Quickstep's overhead examined in the previous paragraph are examples of this.

Tectorwise & Typer have the same performance at the projectivity of degree one. As the projectivity increases, Typer outperforms Tectorwise at single-threaded execution. This is because, as the projectivity increases, Tectorwise suffers more from the materialization overhead. Whereas,

Table 2: Normalized response times for projection microbenchmark for single- and multi-threaded executions.

	Single-threaded				Multi-threaded			
	p1	p2	p3	p4	p1	p2	p3	p4
R	56	39.3	35.5	32.8	43	30.1	24.5	22
C	13.6	14.4	15.4	15.3	9	10.1	10.3	10
Qs	2.7	2.7	2.7	2.8	1.7	1.8	1.7	1.8
V	1	1	1	1	1	1	1	1
Tw	0.5	0.6	0.7	0.8	0.6	0.7	0.7	0.7
Ty	0.5	0.4	0.5	0.5	0.6	0.7	0.7	0.7

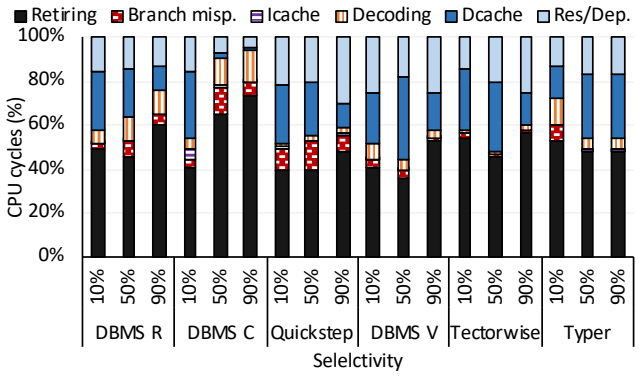

Figure 3: Breakdowns of the CPU cycles for selection microbenchmark for single-threaded execution.

Table 3: Consumed bandwidth in GB/s for projection microbenchmark for single- and multi-threaded executions.

	Single-threaded				Multi-threaded			
	p1	p2	p3	p4	p1	p2	p3	p4
R	0	0	0	0	44.6	34.7	27.3	23.6
C	0	0	0	0	0.9	0.6	0.3	0.3
Qs	0	0.1	0.9	0.7	21.6	23.5	24	24.2
V	2.8	2.6	2.4	2.2	38.6	43.2	43.8	45.6
Tw	7	6.8	5	4.9	62.9	62.4	61.5	61.1
Ty	8.6	10.5	9.6	10.1	62.8	63	62.9	62.8

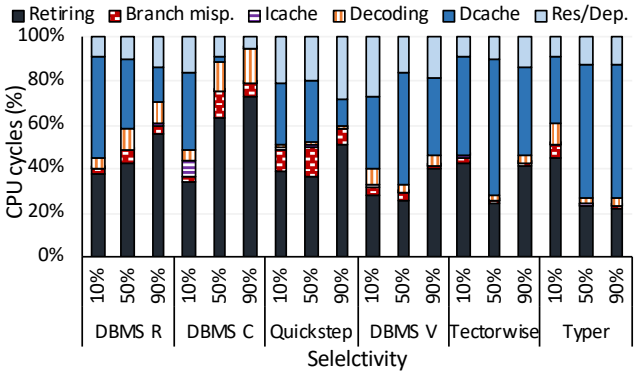

Figure 4: Breakdowns of the CPU cycles for selection microbenchmark for multi-threaded execution.

Typer follows a compiled execution model that does not suffer from the materialization overhead.

Typer's and Tectorwise's relative performances are the same at the multi-threaded execution, as they are both memory-bandwidth bound. Table 3 shows that both Tectorwise and Typer saturate the memory bandwidth at the multi-threaded execution. We also examine Typer's and Tectorwise's function-call traces. They both spend almost 100% of their time inside the aggregation function.

Single vs. Multi-threaded Execution: DBMS C and Quickstep have the same breakdowns of the CPU cycles for the single- and multi-threaded executions. Table 3 shows that DBMS C has very low single- and multi-threaded bandwidth consumption, which explains the similar micro-architectural behavior. Quickstep has a low single-threaded, yet significant multi-threaded bandwidth consumption. Nevertheless, its bandwidth stress is not sufficiently high to change the micro-architectural behavior.

DBMS V's, Tectorwise's and Typer's have high Dcache stalls. DBMS V consumes a large fraction of the memory bandwidth, which results in higher Dcache stalls. Whereas, Tectorwise and Typer fully consume the memory bandwidth, which results in highly pronounced Dcache stalls.

4. SELECTION

We present the selection microbenchmark. Figure 3 and 4 show the breakdowns of the CPU cycles, Table 4 shows normalized response times, and Table 5 shows consumed memory bandwidths for single- and multi-threaded executions. We use the highest performing version of the branched vs. branch-free implementations for Tectorwise and Typer. This version is the branched version for Typer at 10% selectivity and the branch-free version for all the other cases.

DBMS R is 13.1 to 33.1 times slower than DBMS V. It spends 50% of its CPU cycles retiring instructions, which highlights its instruction overhead.

DBMS C is 40% faster than DBMS V and consumes a large amount of memory bandwidth at 10% selectivity. As the selectivity increases, DBMS C becomes significantly slower and consumes increasingly less bandwidth.

DBMS C keeps its columns in 1MBs of blocks together with some metadata information per block such as min., max. and count values. In the case of low selectivities, it simply scans the metadata and skips the blocks that are not necessary. Hence, it provides high performance. We confirm our hypothesis by using microbenchmarks that use only metadata information and by obtaining similar results.

Quickstep is 3.4 to 5.5 times slower than DBMS V. It spends 40% of its CPU cycles retiring instructions, and 20% for Dcache stalls. This highlights its instruction and data overhead. To understand the instruction overhead, we examine Quickstep's function-call trace. Quickstep spends 40.8% of its time in `BitVector::firstOne()`, and 15.5% in `BitVector::setBit()`. Quickstep uses these two functions to avoid processing the tuples that are already filtered out by the previous predicates in a conjunctive condition. `BitVector::firstOne()` relies also on a C++ construct `__builtin_clz()` to count leading zero bits of an integer. As the predicate evaluation, by itself, is a simple condition-check operation, a library function-call, together with bitvector manipulations, takes a large fraction of Quickstep's time on selection processing.

Table 4: Normalized response times for selection microbenchmark for single- and multi-threaded executions.

	Single-threaded			Multi-threaded		
	10%	50%	90%	10%	50%	90%
R	33.1	18.9	18.5	21.1	13.1	14.6
C	0.6	3.9	10.9	0.7	2.6	8.2
Qs	7.1	5.5	6.4	3.4	3.4	4.4
V	1	1	1	1	1	1
Tw	0.7	0.7	0.7	0.4	0.7	0.7
Ty	0.8	0.7	0.4	0.4	0.9	0.7

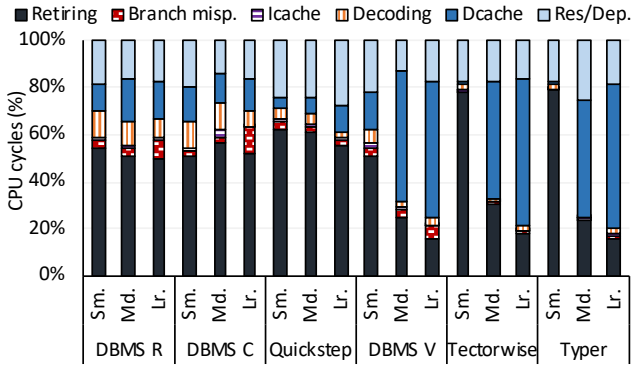

Figure 5: Breakdowns of the CPU cycles for join microbenchmark for single-threaded execution.

Tectorwise uses selection vectors to avoid processing already filtered tuples. A selection vector keeps the IDs of the tuples that should be evaluated for the second and onwards predicates. As selection vectors require a single cache-resident lookup, they are likely to be more efficient than bitvectors. Quickstep could benefit from it.

The breakdown of Dcache stalls shows that ∼70% of the stalls are due to 4K Aliasing that comes from `DateLit:<` operator, which is used for date comparison. 4K Aliasing occurs when the memory addresses of successive load and store operations are aliased by 4K. In this case, hardware fails to perform the *store-to-load forwarding* optimization. This causes a five-cycle penalty and can be significant if it happens frequently. 4K Aliasing can be solved by aligning the data blocks to 32 bytes, or by changing offsets between input and output buffers [13].

DBMS V & Tectorwise have a 30% performance gap, except for 10% selectivity at the multi-threaded execution. DBMS V scales the worst at 10% selectivity. This is likely due to the scalability limitations of the exchange operator that DBMS V relies on. The exchange operator statically creates a number of producer and consumer threads and, in the case of uneven distribution of the tuples, suffers from load imbalance. As, at lower selectivities, the uneven load is likely higher, DBMS V scales worse at lower selectivities. Tectorwise uses morsel-driven parallelism that scales better under uneven loads [21].

We also examine Typer's and Tectorwise's function-call traces. They both spend almost 100% of their time inside the filter and aggregation functions. Their codebases are efficiently implemented, without any instruction overhead.

Table 5: Consumed bandwidth in GB/s for selection microbenchmark for single- and multi-threaded executions.

	Single-threaded			Multi-threaded		
	10%	50%	90%	10%	50%	90%
R	0.7	0	0	46	36.4	23.2
C	3	0	0	34	12.8	2.6
Qs	0	0	0	9.2	18.3	10.8
V	0.6	1.9	1.9	12.6	44.6	39.7
Tw	3	6.8	4.7	50.4	62.7	58.7
Ty	3	9	8.4	51.7	62.8	62.8

Table 6: Normalized response times for join microbenchmark for single- and multi-threaded executions.

	Single-threaded			Multi-threaded		
	Sm.	Md.	Lr.	Sm.	Md.	Lr.
R	7.2	6.8	6.1	2.0	5.3	4
C	7.8	3.8	4.8	2.1	4.5	3.5
Qs	1.9	1.7	1.1	0.4	1.3	0.8
V	1	1	1	1	1	1
Tw	0.2	0.7	0.6	0.1	0.4	0.5
Ty	0.3	0.7	0.6	0.1	0.5	0.5

Single vs. Multi-threaded Execution: At 10% selectivity, DBMS C has high Dcache stalls and bandwidth consumption, due to its fast meta-data processing technique. Quickstep's micro-architectural behavior is the same for single- and multi-threaded executions, as its instruction overhead prevents it from stressing the memory bandwidth. Due to their high bandwidth stress, DBMS V, Tectorwise, and Typer all significantly suffer from Dcache stalls at the multi-threaded execution for all the selectivities.

5. JOIN

We present the join microbenchmark. Figure 5 shows the breakdowns of the CPU cycles for single-threaded execution. Table 6 shows normalized response times, and Table 7 shows consumed memory bandwidths for single- and multi-threaded executions. We omit the breakdowns of the CPU cycles for multi-threaded execution as it is the same as that of single-threaded. All systems use the hash join algorithm.

DBMS R & C are 2 to 7.2 times slower than DBMS V. They spend the majority of the CPU cycles retiring cycles, which points to their instruction overhead.

Quickstep is 10% slower and 20% faster than DBMS V for single- and multi-threaded executions for the large-sized join. The reason is that Quickstep converts the hash join into a *Filter Joins (FJ)* if (i) the probe-side join key is unique, and (ii) if no attribute is required from the probe side in the result of the join. In this case, FJ builds an *Exact Filter (EF)*, rather than a hash table, on the build side. An EF is a bitvector where every build key corresponds to a single bit. FJ then probes the EF to decide whether a tuple from the probe side should pass the join.

In Figure 6, we examine the breakdown of the normalized response-time at the hardware- (left) and software-levels (right), with and without using FJ. We make a best effort categorization of Quickstep's methods. To illustrate this, the intermediate-result materialization category (Int. res.

Table 7: Consumed bandwidth in GB/s for join microbenchmark for single- and multi-threaded executions.

	Single-threaded			Multi-threaded		
	Sm.	Md.	Lr.	Sm.	Md.	Lr.
R	0	0	0	2.6	30.2	12.1
C	0	0	0	0.4	18.6	3.1
Qs	0	0	0	0	9.6	13.8
V	0	1	0.9	0	8.5	17.3
Tw	0	0	1.3	0	15.4	23.1
Ty	0	0	1.2	0	11.5	21.3

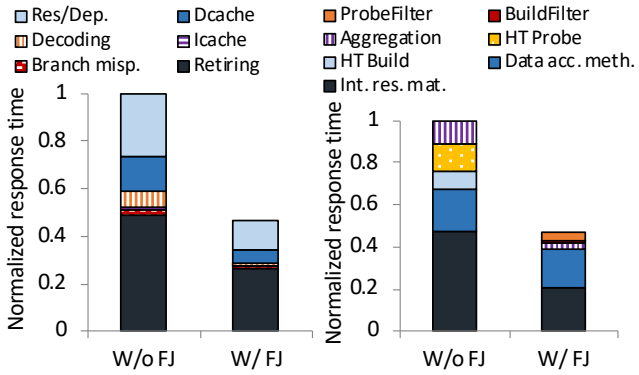

Figure 6: Normalized response time breakdowns at the hardware- (left) and software-levels (right) for Quickstep when it runs the large join microbenchmark query, as single-threaded, with and without using Filter Join (FJ).

mat) includes methods such as `bulkInsertTuplesWithRemappedAttributes()` and `appendUntypedValue()`. Data-access methods (Data acc. meth.) include methods such as `getUntypedValue()`, `getTypedValue()`, and `next()`.

FJ improves the response time by ∼50%. Quickstep does not suffer from Dcache stalls, even without FJ. This shows the overhead that Quickstep carries. Quickstep spends (with and without FJ) half of its time on intermediate-result materialization. Late materialization is known to be a major performance bottleneck for OLAP systems [1].

DBMS V, Tectorwise & Typer spend the majority of their time on Dcache stalls for both middle- and large-sized joins. Hence, hash join is Dcache-stalls bound even for the middle size, whose build-side table (supplier) is 700K tuples.

We also examine Typer's and Tectorwise's function-call traces. They both spend almost 100% of their time inside the join and aggregation functions. Their codebases are efficiently implemented, without instruction overhead.

Build vs. Probe Phases: Hash join is composed of two phases: build and probe. Both phases compute, based on a key, a hash value. They then make a hash table lookup for an insertion purpose while building, and for a read purpose while probing. We observe that the micro-architectural behavior of both build and probe phases are largely Dcache-stalls dominated. Hence, the random-data accesses dominate both phases. We omit the breakdowns of the CPU cycles in the interest of space.

Probe phase dominates the execution time for single-threaded execution (61% of the time), while build phase dominates the execution time for multi-threaded execution (53% of the time), both for Typer and Tectorwise. This is due to the scalability bottlenecks of build phase that relies on atomic exchange instructions for concurrent hash-table inserts.

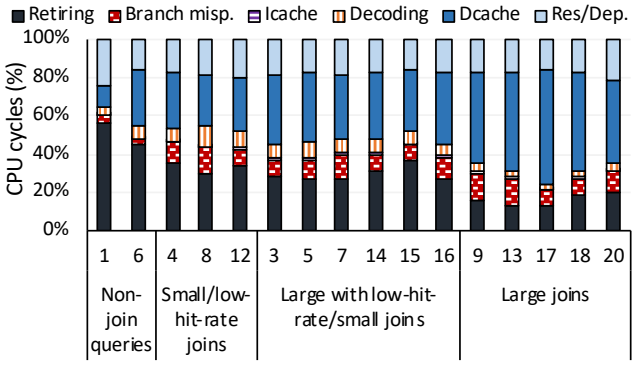

Figure 7: Breakdowns of the CPU cycles for a large subset of TPC-H queries for single-threaded execution.

Single vs. Multi-threaded Execution: Multi-threaded execution does not change micro-architectural behavior. This is because none of the systems is able to sufficiently stress the memory bandwidth. Table 7 shows that the maximum consumed bandwidth is 23.1GB/s; this is well below the maximum random-data access bandwidth of 60GB/s. Hence, when the hash join algorithm is running, the memory bandwidth is largely underutilized . We omit the breakdowns of the CPU cycles for the multi-threaded execution in the interest of space.

By this finding, we confirm the existing work that uses co-routines to improve hash join performance [14, 25]. Co-routines enable the overlapping of long-latency memory stalls with computation for a more efficient utilization of the memory bandwidth. Psaropoulos et al. [26, 27] show that memory bandwidth starts being saturated with 28 or more cores.

6. TPC-H

We present a TPC-H benchmark evaluation. We first profile a large subset of TPC-H queries when they are run by DBMS V. We then choose six representative queries, and continue with the cross-system comparison.

We identify two main dimensions in the categorization of the TPC-H queries: join size and hit rate. Join size is defined by the size of the probe-side hash table, as it defines how cache-resident the hash join is. Hit rate is defined by the number of times that the probe side finds a matching entry at the build side. If this value is less than 10%, we identify the join as a low hit-rate join. Low hit-rate enables us to use bloom filters to reduce the number of hash probes [6]. The smaller the join size is and the lower the hit rate is, the less the Dcache stalls the join suffers from.

Figure 7 presents the breakdowns of the CPU cycles for single-threaded execution. Based on their micro-architectural behavior, there are four main classes of queries. Q1 and Q6 are the non-join queries and have relatively high retiring-cycles ratios. Q4, 8 and 12 are queries with small-sized joins or large-sized joins, both with low hit-rates. They suffer from Dcache stalls at ∼25%. Q3, 5, 7, 14, 15 and 16 are

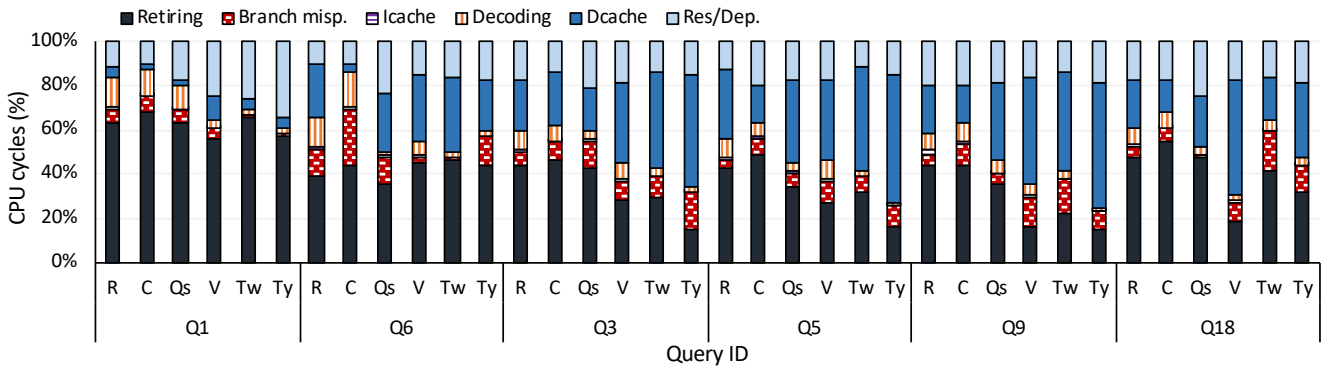

Figure 8: Breakdowns of the CPU cycles for TPC-H Q1, Q6, Q3, Q5, Q9 and Q18 for single-threaded execution.

Table 8: Normalized response times for TPC-H queries for single-threaded execution.

	Q1	Q6	Q3	Q5	Q9	Q18
R	25.7	16.7	9.3	10.4	4.7	2.7
C	21	4.5	3.4	5	3.4	1.9
Qs	5	6.4	1.8	4.6	1.8	0.3
V	1	1	1	1	1	1
Tw	1.1	0.6	1.1	1	0.7	0.3
Ty	0.7	0.7	1.5	1.4	0.9	0.3

Table 9: Normalized response times for TPC-H queries for multi-threaded execution.

	Q1	Q6	Q3	Q5	Q9	Q18
R	23.1	16.7	4.9	11.6	4.8	1.9
C	18.9	4.1	2.3	6.5	6.7	1.2
Qs	4.4	4.9	0.8	4.3	1.4	0.3
V	1	1	1	1	1	1
Tw	1	0.6	0.4	0.9	0.6	0.3
Ty	0.6	0.7	0.5	1.1	0.7	0.3

queries with large joins mixed with low hit-rate large joins or small joins. These queries suffer from Dcache stalls at ∼35%. Lastly, Q9, 13, 17, 18 and 20 are joins with large sizes with high hit-rates. These queries suffer from Dcache stalls at ∼45%.

We choose Q1, 6, 3, 5, 9 and 18 to continue with the cross-system evaluation. Figure 8 shows the breakdowns of the CPU cycles for single-threaded execution. Table 8 and 9 show normalized response times, and Table 10 and 11 show consumed bandwidth values for single- and multi-threaded executions. We present the breakdowns of the CPU cycles for multi-threaded execution only for Q6 in Figure 9, as they are the same as the single-threaded breakdowns for all the other queries and systems.

Q1 is an aggregation-heavy query with high temporal-locality. All the systems have high retiring-cycles ratios. DBMS R and C are 25.7 and 21 times slower than DBMS V, which highlights their instruction overhead. Quickstep is 5 times slower than DBMS V, due to its aggregation overhead (shown in Section 3). DBMS V and Tectorwise have close performances, which shows that DBMS V's real-life system overhead is compensated in an aggregation-heavy query. Typer is 30% faster than DBMS V and Tectorwise, as Typer does not suffer from materialization overhead.

Q6 is a scan-intensive query that scans three columns and evaluates five predicates over them. We use Typer's branched and Tectorwise's branch-free versions. As the selectivity of the query is very low (1.9%), most of the time is spent in predicate evaluations.

Due to its instruction and data overhead, DBMS R is 16.7 times slower than DBMS V. DBMS C is 4.1 times slower than DBMS V with instruction overhead. Quickstep suffers from its selection operator overhead that is similar to the

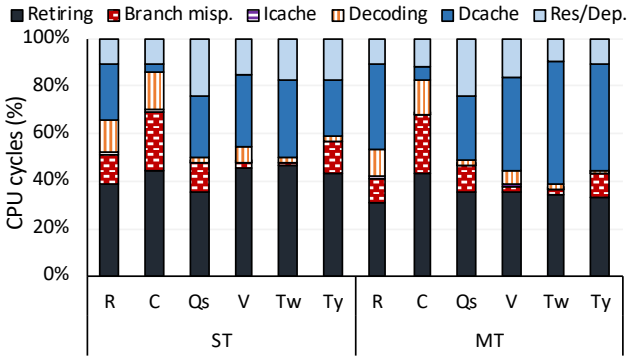

Figure 9: Breakdowns of the CPU cycles for TPC-H Q6 for single- (ST) vs. multi-threaded (MT) executions.

selection microbenchmark. It is 4.9 times slower than DBMS V. It suffers from Dcache stalls due to 4K Aliasing.

DBMS V, Tectorwise, and Typer have high retiring cycles at single-threaded execution, but significantly suffer from Dcache stalls at multi-threaded execution (see Figure 9). Because all three systems approach the bandwidth limits (see Table 11). DBMS C's and Quickstep's breakdowns of the CPU cycles are similar at single- and multi-threaded executions due to their low bandwidth stresses.

Q3 is a join-intensive query where three large tables of TPC-H, lineitem, orders and customer are joined. DBMS R and C are significantly slower than DBMS V. Quickstep is 1.8 times slower and 20% faster than DBMS V, at the single- and multi-threaded executions. Quickstep implements two important join optimizations: Filter Joins (FJ) and Lookahead

Table 10: Consumed bandwidth in GB/s for TPC-H queries for single-threaded execution.

	Q1	Q6	Q3	Q5	Q9	Q18
R	0	0	0	0	0	0
C	0	0	0	0	0.1	0.1
Qs	0	0	0	0	0	0.2
V	0.4	2.2	0.4	0.1	0.4	1
Tw	0	5	0.9	0.2	1.1	0.6
Ty	1	5	0.1	0.1	0.3	1.3

Table 11: Consumed bandwidth in GB/s for TPC-H queries for multi-threaded execution.

	Q1	Q6	Q3	Q5	Q9	Q18
R	7.4	41.9	30.4	22.8	21.3	24.2
C	0	5.9	12.8	11.2	8.2	7.4
Qs	4.9	6.7	8.5	7.6	7.8	10.4
V	22.9	40.8	12.1	9.7	16.2	21.7
Tw	18.9	56.2	22.4	20.7	27.7	17.8
Ty	29.1	57.9	16.7	14.4	21.9	21.1

Information Passing (LIP) filters. FJ replaces the build-side hash table with a bitvector, as explained in Section 5. LIP filters are bloom filters that are passed down in the query plan so that a join can drop rows that would satisfy the current join, but not a future join.

We turn on/off FJ and LIP filter, and we measure how useful they are for Q3. FJ improves Q3's response time by 47%, and LIP filters further improve it by 28%; overall, providing a 65% reduction in the response time. Despite eliminating the major costs of hash joins in Q3, Quickstep still spends a significant amount of time on retiring instructions, which shows that it nevertheless suffers from overhead that is identified in Section 3, 4 and 5.

Tectorwise and Typer are slower than DBMS V, because DBMS V, as the hit rate is low (1%), uses bloom filter on its join with lineitem and the result of the orders and customer join. To test our hypothesis, we perform a subquery analysis for Q3 by excluding, and then including, the lineitem join. We see that the Dcache stalls are DRAM-dominated without the lineitem join, whereas L2- and L3-dominated with the lineitem join, which supports our conclusion.

Tectorwise is faster than Typer. Tectorwise separates hash computing from hash probing by saving computed hashes in an intermediate vector. This enables the overlapping of costly random-data accesses at the hash probing phase. Typer, on the other hand, performs hash computation and hash probing one-after-the-other. It also combines the filtering condition and hash table probe operation in a single if condition. This mixes the random-data access further with a sequential scan of a column, which is used to filter the data (such as: `if(o_orderdate[i] < c1 & ht1.contains( o_custkey[i]))`). As a result, Typer is not able to overlap the random-data accesses as much as Tectorwise does. This shows that materialization overhead of the vectorized engine pays off for the hash join operation unlike the case for projection and selection. DBMS V, Tectorwise and Typer are all Dcache-stalls dominated in their execution time, due to hash join's large number of random-data accesses.

Q5 is a join-intensive query similar to Q3. DBMS R and C are 10.4 and 5 times slower than DBMS V. Quickstep is 4.6 times slower than DBMS V, as Q5 is not able to benefit from FJ and only partially benefits from LIP filters (by a 30% decrease in the execution time).

DBMS V and Tectorwise have a comparable performance, as DBMS V uses a bloom filter while joining the lineitem table. The join's hit-rate of 3%. Similarly, DBMS V spends less time on Dcache stalls compared to Tectorwise. Tectorwise is faster than Typer as Tectorwise can overlap random-data accesses. Nevertheless, all three high-performance systems spend the majority of their time on Dcache stalls.

Q9 is a join-intensive query with large joins and high hit-rates. DBMS R and C are 4.7 and 3.4 times slower than DBMS V. Quickstep is 1.8 times slower than DBMS V. LIP filters improve Q9's time by 55% thanks to filtering out lineitem tuples in its join with the sub-tree of joins of partsupp, part and supplier.

DBMS V is not able to benefit from bloom filters for Q9 as the hit rate is high for all the joins. Hit rate is 20% for the join between the orders and lineitem table, and 50% for the join between the partsupp table and the sub-tree of joins of the rest of the tables. Tectorwise is once again faster than Typer thanks to its random-data access overlapping ability. DBMS V, Tectorwise and Typer all spend the majority of their time on Dcache stalls as Q9 is join-intensive.

Q18 contains a large group by on the lineitem table based on `l_orderkey`, without any filter on top of the lineitem table. It creates 105M groups out of the 420M-sized lineitem table (the scaling factor is 70).

DBMS V is 3.3x times slower than Tectorwise and Typer. Because DBMS V's query optimizer produces a sub-optimal plan. The optimal plan does a single group by over the lineitem table, filters it based on the `having` condition, and feeds the result into a series of joins with orders, customer and another lineitem. DBMS V, however, is not able to push the large group by down the tree. It requires a full join among the lineitem, orders and customer tables, which it further joins with the filtered group by. This results in high response time, together with high Dcache stalls.

DBMS R and C are 2.7 and 1.9 times slower than DBMS V and spend 40% of their time on retiring instructions. Quickstep is as fast as Tectorwise and Typer. Q18 benefits from LIP filters by 30%. It does not suffer from the intermediate-result materialization, as the intermediate results are small (due to the `having` condition) for Q18.

Tectorwise and Typer have relatively high retiring-cycles ratios. Typer does thread-local pre-aggregations for each morsel. Then, Typer globally combines the local pre-aggregations for the final group by. Similarly, Tectorwise creates local groups per vector. Then, it combines the groups globally at the end. Hence, they both work with smaller-sized hash tables that are more cache-resident, which results in high retiring-cycles ratios.

7. MIXED QUERY WORKLOAD

We present a mixed query workload evaluation. Scan-intensive queries are bandwidth-bounded, hence do not scale after a certain number of cores. Join-intensive queries do not create enough memory traffic, hence leave the bandwidth underutilized. In this section, we concurrently run a scan- and join-intensive query, where we create enough memory traffic and also use all the cores on the chip.

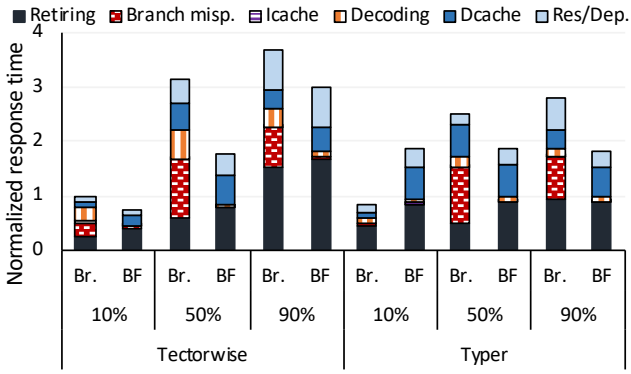

Figure 10: Normalized response time breakdowns for predication for single-threaded execution.

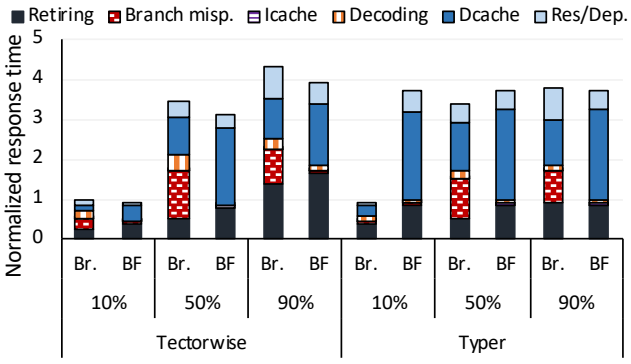

Figure 11: Normalized response time breakdowns for predication for multi-threaded execution.

Table 12: Normalized response times for mixed query workload evaluation.

	Proj.+Q3		Q6+Q3	
	Proj.	Q3	Q6	Q3
Qs	1	1.1	1	1
V	1.1	1.6	1.1	1.5
Ty	1.1	1.9	1.0	1.4

We examine the following two scenarios: (i) projection microbenchmark query with the degree of four running with TPC-H, Q3, and (ii) TPC-H, Q6 running with TPC-H, Q3. We use Quickstep, DBMS V and Typer to evaluate the concurrency scenarios. We choose these three systems, as they stress the memory bandwidth at different levels.

We use eight threads for projection and Q6, and use six threads for Q3 on Typer, as projection does not scale after eight cores on Typer. We use ten threads for projection and Q6, and use four threads for Q3 on DBMS V, as Q3 does not scale after four cores on DBMS V[1]. We use the same configuration for Quickstep as for Typer.

Table 12 presents concurrent response times normalized to the corresponding non-concurrent response times. For DBMS V and Typer, Q3's response time is significantly higher when it runs with the projection and TPC-H, Q6, whereas, for Quickstep, Q3's response time does not change signficiantly. Because DBMS V and Typer sufficiently stress the memory bandwidth to interfere with Q3, while Quickstep does not. On the other hand, projection's and Q6's response times do not change significantly. Hence, concurrent execution enables us to use the underutilized cores left by the scan-intensive query. However, it causes interference in the shared memory bandwidth, hence results in a decreased execution time for the join-intensive query.

We also examine the micro-architectural behavior. The results are as expected. Q3's Dcache stalls are increased substantially when running with projection/Q6 on Typer and DBMS V, whereas remain the same on Quickstep. The total consumed bandwidth of a concurrent execution is the sum of the individual bandwidth consumptions unless the sum reaches to the maximum bandwidth. We omit the graph and table for these for brevity.

[1]We microbenchmarked Q3 on DBMS V by varying the selectivity of the filter on the lineitem table from 100% to 50%. We realized that Q3 starts not scaling when the selectivity drops below 70%. This suggests that the reason for not scaling is the load imbalance issue of the exchange operator that creates uneven loads at lower selectivities.

8. PREDICATION

We present predication evaluation. Listing 1 presents an example of predication. Line 1 to 3 present regular branched execution, whereas Line 4 to 6 present the predicated execution. Predication takes the conditional expression out of the `if()` statement and assigns the expression to the variable `decision`. Then, it uses the `decision` variable to compute the final result. If the `decision` is zero, it will not affect the final result, whereas if the `decision` is one, it will update the final result as in the branched execution. Predication requires more computation but allows for avoiding costly branch mispredictions.

```
1  // branched version
2  if( (a < v1) & (b < v2) & (c < v3) )
3      result += (d + e);
4  // branch-free, predicated version
5  bool decision = (a < v1) & (b < v2) & (c < v3);
6  result += (decision * (d + e));
```

Listing 1: Predication example.

The conditional expression uses bitwise and as opposed to logical and. Compiler generates a branch instruction for each logical and. Hence, a logical and triggers branch predictor even if it is not in an `if()` statement. However, bitwise and translates into a set of bitwise operations followed by a single conditional branch, which can be eliminated by taking the expression out of the `if()` statement.

Figure 10 shows normalized response time breakdowns for single-threaded execution where all values are normalized to the left-most bar. We see that branch mispredictions are a major source of cost. Predication reduces the response time for all the cases except for Typer at 10% selectivity.

Typer improves performance the highest at 90%, although branch misprediction cost is the highest at 50%. This is because the computation overhead that predication brings is less at 90% compared to 50%, as the unpredicated query computes the aggregation for most of the tuples at 90%.

Typer suffers significantly from branch mispredictions at 90% selectivity. Because Typer uses bitwise and to implement the conjunction, hence suffers from the overall selectivity of the conjunction rather than the individual predicate

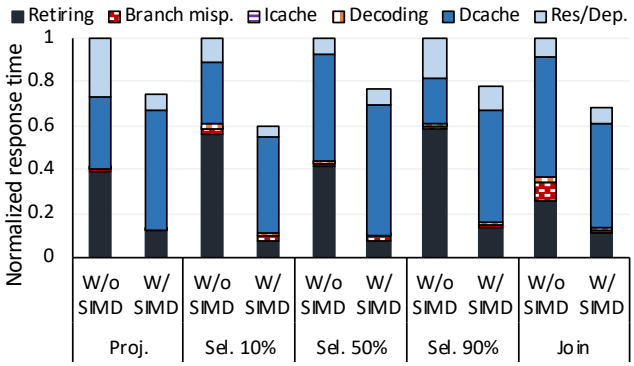

Figure 12: Normalized response time breakdowns for SIMD for single-threaded execution.

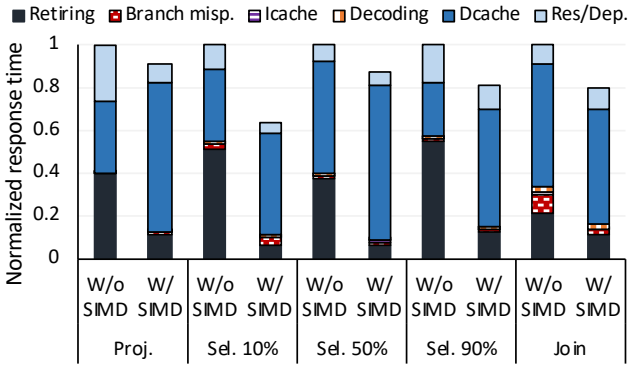

Figure 13: Normalized response time breakdowns for SIMD for multi-threaded execution.

Table 13: Consumed bandwidth in GB/s for SIMD for single-threaded execution.

	Proj.	Sl.10%	Sl.50%	Sl.90%	Join
W/o SIMD	6	4	8	6	2.8
W/ SIMD	8	8	8	8	4.5

Table 14: Consumed bandwidth in GB/s for SIMD for multi-threaded execution.

	Proj.	Sl.10%	Sl.50%	Sl.90%	Join
W/o SIMD	71.4	49.9	76.2	63.7	34.8
W/ SIMD	79.4	73.1	81.4	79.2	40.4

selectivities. In this case, it is $90\% \times 90\% \times 90\% = 73\%$, which is less predictable than 90%.

Figure 11 presents normalized response time breakdowns for multi-threaded execution. It shows that the majority of the performance gains are lost due to bandwidth limitations for all the queries. Hence, while predication can significantly reduce the response time, its multi-core benefits are limited by the maximum memory bandwidth.

We also profiled predicated TPC-H, Q6 on Typer and Tectorwise, and we reached similar conclusions. We omit the graphs for Q6.

9. SIMD

We present SIMD evaluation. SIMD instructions perform multiple arithmetic/logic operation in a single instruction. We test Tectorwise when running the projection, selection and join microbenchmarks, with and without using SIMD. As our Broadwell server does not support AVX-512 instructions, we do all the SIMD experiments on a Skylake server supporting AVX-512 instructions.

The Skylake server has a different memory hierarchy than that of the Broadwell server. As a result, the reported values that do not use SIMD do not exactly match with the values reported earlier in the paper (see Section 2, Hardware subsection for more details).

9.1 Projection & Selection

Figure 12 shows the normalized response time breakdowns. Table 13 presents single-core bandwidth consumption values. We use the predicated, branch-free versions of the selection queries as SIMD is more effective when branch mispredictions are eliminated. The figure shows that there is a 70% to 87% decrease in the amount of time spent for retiring cycles for all four cases. As retiring cycles are correlated to the number of retired instructions, SIMD successfully reduces the number of retired instructions.

While the number of retired instructions is reduced, Dcache stalls are increased by 20% to 2.3x. Hence, overall SIMD gains are limited by the increased Dcache stalls. Table 13 shows that the projection and selection queries are single-core-bandwidth bound when using SIMD, as the maximum per-core bandwidth is 8GB/s. Hence, per-core SIMD gains are limited by per-core bandwidth.

Figure 13 shows normalized response time breakdowns, and Table 14 presents the bandwidth consumption for multi-threaded execution. SIMD gains are less at the multi-th-eaded execution, due to approaching the bandwidth limits.

We also run projection and predicated selection without SIMD on Typer on the Skylake server. Typer saturates the per-core and multi-core bandwidth and does not scale after 8 cores. Hence, its SIMD gains would be less than that of Tectorwise that, without SIMD, is not able to saturate per-core or per-socket bandwidths.

9.2 Join

Figure 12 shows that SIMD significantly reduces the number of retired instruction and does not increase the Dcache stalls. Table 13 shows that single-core bandwidth consumption is well below the maximum (7GB/s), without and with SIMD. Hence, SIMD is able to exploit the available core bandwidth and reduce the response time, without increasing the Dcache stalls time. SIMD gains are less pronounced at the multi-threaded execution, due to the stress on the memory bandwidth. As Table 14 shows, join consumes 40GB/s of the 60GB/s random access bandwidth.

10. HARDWARE PREFETCHERS

We present the hardware prefetcher evaluation. We study four hardware prefetchers that today's server processors provide: L1 next line (L1 NL), L1 streamer (L1 Str.), L2 next line (L2 NL) and L2 streamer (L2 Str.) prefetchers. We turn on each prefetcher individually and examine its effect on the micro-architectural behavior.

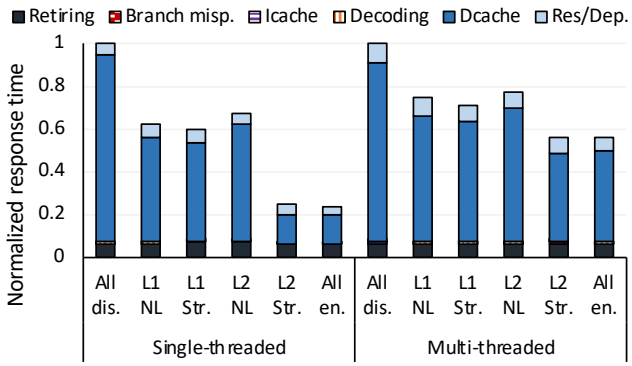

Figure 14: Normalized response time breakdowns for hardware prefetcher for single- and multi-threaded executions.

Figure 14 shows the normalized response time breakdowns, and Table 15 shows the consumed bandwidth values when running the projection query of degree four on Typer for single- and multi-threaded executions.

Prefetchers reduce the response time by 75% for single-threaded execution. This shows that hardware prefetchers are highly useful for a sequential-scan-heavy query. For multi-threaded execution, prefetchers reduce the response time less than single-threaded execution. Hence, the benefits of the prefetchers are limited by the maximum bandwidth.

We also examine the projection on Tectoriwse and the branched and branch-free selection on Typer and Tectorwise. The results agree with our findings for the projection query on Typer. We also examined the join microbenchmark. Prefetchers reduce the response time modestly by $\sim$ 20% for the large-sized join both for Typer and Tectorwise.

11. HYPER-THREADING AND TURBO-BOOST

We present the hyper-threading (HT) and turbo-boost (TB) evaluation. We first compare single-core performance only having TB enabled. We examine DBMS V, Tectorwise and Typer for the projection (of degree four) query, the large join query, branched and branch-free versions of the selection query (at 50% selectivity), TPC-H, Q1, Q6, and Q3. The maximum speedup we observe is by 27% for TPC-H, Q1 that runs on Typer. As Q1 is arithmetic-operation-heavy, and Typer does not suffer from the materialization overhead, turbo-boost worked the best. The other improvements were modest and were 10 to 20%. TB's benefits are less as the number of cores is increased. Hence, the single-core results provide the best case. Hence, TB-alone provides only modest performance improvements.

Next, in addition to TB, we enable HT. We examine how much HT improves the performance. We examine two scenarios: 2H and 28H. 2H shows the response time of using 2 HTs that share a physical core normalized to the response time of using one physical core. 28H shows the response time of using 28 HTs that share 14 physical cores normalized to the response time of using 14 physical cores. All the queries scale well across 2 physical cores.

Table 16 presents the results for a selection of queries we have covered up to now. Proj. is the projection query with the degree of four, and Join is the large-sized join query. 2H

Table 15: Consumed bandwidth in GB/s for hardware prefetcher for single- (ST) and multi-threaded (MT) executions.

	All dis.	L1 NL	L1 S	L2 NL	L2 S	All en.
ST	1	3	3	3	8.4	10.1
MT	32.9	42.4	43.2	41.2	62.5	62.8

reduces the response time maximum by 34%, for Typer when running Q3. As Typer is not able to effectively overlap the random-data accesses at the software-level, HT enables overlapping them. Similarly, branched selection query at 50% selectivity benefits 34% on Tectorwise and 32% on Typer as it can overlap the branch misprediction stalls. For the rest of the cases, 2H improve the performance modestly by 20% on average. 28H improves performance less than that of 2H due to the increased stress on memory bandwidth.

Table 17 shows how useful HTs are for different prefetcher configurations, for the projection (of degree four) and large join query on Typer. The less aggressive the prefetchers are, the more benefits HTs provide. Hence, while prefetchers significantly improve the performance, it limits the benefit of HTs. For join, HT benefits do not change much, as prefetchers are not very useful for join.

Table 18 presents the case for SIMD, for the projection query (of degree four), selection query at 10% selectivity (branch-free), and large join query on Tectorwise (using the Skylake server). It shows that HT is less useful when using SIMD, though the difference usually is not high.

12. LESSONS LEARNED

OLAP systems should first optimize their instruction footprint. The sources of instruction overhead could be legacy codebases, inefficient implementations, or intermediate-result materialization [29].

Projection and selection performance are sensitive to overhead, as they are usually rapid. Having an inefficient data-access method inside the projection operator or using a simple bitvector operation inside the selection operator loop can significantly increase the execution time. Intermediate-result materialization is a major source of overhead and should be avoided. Using selection vectors is a good alternative to using bitvectors and intermediate-result materialization.

Filtering-based techniques, such as Filter Join, Lookahead Information Passing, block-skipping by metadata processing and bloom filters on low hit-rate joins, are promising techniques to reduce the work being done. However, they do not eliminate the instruction overhead. Combining filtering-based techniques with efficient query-operator implementation can both reduce the work being done and eliminate the instruction overhead.

Vectorized engines are faster than compiled engines, only if their materialization cost pays off. The materialization costs pay off for hash join, as vectors of computed hashes enable the overlapping of costly random hash-table accesses, but the costs do not pay off for projection and selection. Compiled engines suffer from mixing random-data accesses with hash computation and/or condition checks, preventing them from overlapping the random-data accesses.

Scan-intensive queries stress the memory bandwidth, which results in earlier saturation of the cores, whereas join-

Table 16: Normalized response times for hyper-threading.

	DBMS V		Twise		Typer	
	2H	28H	2H	28H	2H	28H
Proj.	0.95	1.03	0.82	1.00	0.92	1.00
Join	0.80	0.84	0.78	0.83	0.77	0.82
Q6	0.82	1.03	0.78	0.93	0.76	0.95
Q3	1.03	1.01	0.85	0.90	0.66	0.71
Sel.50%-Br.	-	-	0.66	0.96	0.68	0.94
Sel.50%-BF	-	-	0.83	1.01	0.87	1.01

inten- sive queries saturate the cores, hence leaving the memory bandwidth underutilized. Concurrent execution can provide a scenario where both core and memory resources are fully utilized. However, concurrently executing queries interfere with each other in the shared memory bandwidth, which results in an increased response time for the join-intensive query. Therefore, OLAP systems should carefully schedule their concurrent queries and be aware of potential interference. Isolation mechanisms, such as Intel's Cache Allocation Technology and dynamically disabling and enabling the prefetchers, can be useful to mitigate the interference [22].

SIMD and predication are useful for improving single-threaded performance but, due to bandwidth limitations, they fall short on multi-threaded performance. Hardware prefetchers are essential for high-performance scans but are not so useful for joins. Hyper-threading and turbo-boost are effective for a few particular scenarios but mostly provide modest speedups. Therefore, hardware (software) developers should design hardware (software) based on software (hardware) characteristics for optimal performance.

13. RELATED WORK

There is a large body of work on the micro-architectural analysis of database workloads. Ailamaki et al. [2] and Hardavellas et al. [8] present analytical and transactional workload characterization. Tozun et al. [33, 34] characterize disk-based OLTP systems. Sirin et al. [29] characterize in-memory OLTP systems. Our work complements these studies by characterizing OLAP workloads.

Kersten et al. [18] examine vectorized and compiled OLAP engines without getting deep into the micro-architectural behavior. Sompolski et al. [31] present a comparison between vectorized and compiled engines in terms of particular optimizations, such as predication and SIMD. Our work extends and complements these works in terms of breadth and depth of the analysis.

Ferdman et al. [7] present micro-architectural analysis of a suite of cloud workloads, by concluding that there is a fundamental mismatch among what today's server processors provide and what the cloud workloads demand. Our work agrees with this work, and extends its conclusions to modern OLAP workloads.

Yasin et al. [36] introduce Top-Down Micro-architecture Analysis Methodology (TMAM) deployed by Intel VTune as general-exploration. Sirin et al. [30] improve TMAM which is adopted by Intel VTune in version 3 and onwards.

Yasin et al. [37] analyze cloud workloads. Sridharan and Patel [32] examine the evaluation of workloads on the popular data analysis language R, over a commodity processor.

Table 17: Normalized response times for hyper-threading for different hardware prefetcher configurations on Typer.

	Proj.		Join	
	2H	28H	2H	28H
All dis.	0.66	0.74	0.75	0.80
L1 NL	0.91	0.94	0.78	0.81
L1 Str.	0.92	0.93	0.76	0.81
L2 NL	0.80	0.86	0.75	0.81
L2 Str.	0.93	1.01	0.75	0.81
All en.	0.92	1.00	0.77	0.82

Table 18: Normalized response times for hyper-threading with and without SIMD on Tectorwise.

	Proj.		Sel.10%-BF		Join	
	2H	28H	2H	28H	2H	28H
W/o SIMD	0.72	0.94	0.71	0.76	0.77	0.83
W/ SIMD	0.82	0.96	0.71	0.93	0.74	0.74

Awan et al. [3, 4] present a micro-architectural analysis of Spark. Kanev et. al. [15] present a micro-architectural profiling of scale-out workloads. Our work complements these studies by the analysis of modern OLAP workloads.

14. CONCLUSION

In this work, we have presented a micro-architectural analysis of a set of OLAP systems. We have examined the breakdowns of the CPU cycles, memory bandwidth consumption values and normalized response times. The results show that traditional commercial OLAP systems suffer from their large instruction footprint, which makes them orders of magnitude slower than high-performance columnstores. High-performance columnstores execute tight instruction streams; however, they spend 25 to 82% of their CPU cycles on stalls. Scan-intensive queries suffer from bandwidth saturation, whereas join-intensive queries suffer from long-latency data-cache misses. A concurrent execution of scan- and join-intensive queries can improve the utilization. However, it creates interference in the shared bandwidth, which results in sub-optimal performance.

15. ACKNOWLEDGMENTS

We thank the anonymous reviewers and the members of the DIAS laboratory for their constructive feedback and support throughout this work. We thank Doruk Cetin, Timo Kersten, Georgios Psarapoulous, Stella Giannakopoulou and Bikash Chandra for their useful comments. This project has received funding from the European Union Seventh Framework Programme (ERC-2013-CoG), under grant agreement no 617508 (ViDa), and Swiss National Science Foundation, Project No.: 200021_146407/1 (Workload- and hardware-aware transaction processing).

16. REFERENCES

[1] D. Abadi, P. Boncz, and S. Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., 2013.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. DBMSs on A Modern Processor: Where Does Time Go? In *VLDB*, pages 266–277, 1999.

[3] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. Performance Characterization of In-Memory Data Analytics on a Modern Cloud Server. In *BDCloud*, pages 1–8, 2015.

[4] A. J. Awan, M. Brorsson, V. Vlassov, and E. Ayguade. Micro-Architectural Characterization of Apache Spark on Batch and Stream Processing Workloads. In *BDCloud*, pages 59–66, 2016.

[5] P. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by A Relational Engine. In *SIGMOD*, pages 479–490, 2006.

[6] P. Boncz, T. Neumann, and O. Erling. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *TPCTC*, pages 61–76, 2013.

[7] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *ASPLOS*, pages 37–48, 2012.

[8] N. Hardavellas, I. Pandis, R. Johnson, N. Mancheril, A. Ailamaki, and B. Falsafi. Database Servers on Chip Multiprocessors: Limitations and Opportunities. In *CIDR*, pages 79–87, 2007.

[9] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[10] Intel. Disclosure of Hardware Prefetcher Control on Some Intel Processors. https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[11] Intel. Intel Memory Latency Checker. https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[12] Intel. Understanding How General Exploration Works in Intel VTune Amplifier, 2018. https://software.intel.com/en-us/articles/understanding-how-general-exploration-works-in-intel-vtune-amplifier-xe.

[13] Intel. Intel(R) 64 and IA-32 Architectures Optimization Reference Manual, 2019.

[14] C. Jonathan, U. F. Minhas, J. Hunter, J. Levandoski, and G. Nishanov. Exploiting Coroutines to Attack the "Killer Nanoseconds". *PVLDB*, 11(11):1702–1714, 2018.

[15] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei, and D. Brooks. Profiling A Warehouse-scale Computer. In *ISCA*, pages 158–169, 2015.

[16] M. Karpathiotakis, I. Alagiannis, and A. Ailamaki. Fast Queries over Heterogeneous Data Through Engine Customization. *PVLDB*, 9(12):972–983, 2016.

[17] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011.

[18] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. Boncz. Everything You Always Wanted to Know About Compiled and Vectorized Queries but Were Afraid to Ask. *PVLDB*, 11(13):2209–2222, 2018.

[19] T. Lahiri, S. Chavan, M. Colgan, D. Das, A. Ganesh, M. Gleeson, S. Hase, A. Holloway, J. Kamp, T. Lee, J. Loaiza, N. Macnaughton, V. Marwah, N. Mukherjee, A. Mullick, S. Muthulingam, V. Raja, M. Roth, E. Soylemez, and M. Zait. Oracle Database In-Memory: A Dual Format In-memory Database. In *ICDE*, pages 1253–1258, 2015.

[20] P.-A. Larson, C. Clinciu, E. N. Hanson, A. Oks, S. L. Price, S. Rangarajan, A. Surna, and Q. Zhou. SQL Server Column Store Indexes. In *SIGMOD*, pages 1177–1184, 2011.

[21] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*, pages 743–754, 2014.

[22] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, pages 450–462, 2015.

[23] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[24] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A Data Platform Based on the Scaling-Up Approach. *PVLDB*, 11(6):663–676, 2018.

[25] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *PVLDB*, 11(2):230–242, 2017.

[26] G. Psaropoulos, T. Legler, N. May, and A. Ailamaki. Interleaving with Coroutines: A Systematic and Practical Approach to Hide Memory Latency in Index Joins. *The VLDB Journal*, Dec 2018.

[27] G. Psaropoulos, I. Oukid, T. Legler, N. May, and A. Ailamaki. Bridging the Latency Gap between NVM and DRAM for Latency-bound Operations. In *DAMON*, pages 13:1–13:8, 2019.

[28] V. Raman, G. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Mueller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. Storm, and L. Zhang. DB2 with BLU Acceleration: So Much More Than Just a Column Store. *PVLDB*, 6(11):1080–1091, 2013.

[29] U. Sirin, P. Tözün, D. Porobic, and A. Ailamaki. Micro-architectural Analysis of In-memory OLTP. In *SIGMOD*, pages 387–402, 2016.

[30] U. Sirin, A. Yasin, and A. Ailamaki. A Methodology for OLTP Micro-architectural Analysis. In *Damon*, pages 1:1–1:10, 2017.

[31] J. Sompolski, M. Zukowski, and P. A. Boncz. Vectorization vs. Compilation in Query Execution. In *Damon*, pages 33–40, 2011.

[32] S. Sridharan and J. M. Patel. Profiling R on A Contemporary Processor. *PVLDB*, 8(2):173–184, 2014.

[33] P. Tözün, B. Gold, and A. Ailamaki. OLTP in Wonderland: Where Do Cache Misses Come From in Major OLTP Components? In *Damon*, page 8, 2013.

[34] P. Tözün, I. Pandis, C. Kaynak, D. Jevdjic, and A. Ailamaki. From A to E: Analyzing TPC's OLTP

Benchmarks: The Obsolete, The Ubiquitous, The Unexplored. In *EDBT*, pages 17–28, 2013.

[35] TPC. Transcation Processing Performance Council. http://www.tpc.org/.

[36] A. Yasin. A Top-Down Method for Performance Analysis and Counters Architecture. In *ISPASS*, pages 35–44, 2014.

[37] A. Yasin, Y. Ben-Asher, and A. Mendelson. Deep-dive Analysis of The Data Analytics Workload in CloudSuite. In *IISWC*, pages 202–211, 2014.