Highlighting the Performance Diversity of Analytical Queries using VOILA

Tim Gubner
tim.gubner@cwi.nl
CWI
The Netherlands

Peter Boncz
boncz@cwi.nl
CWI
The Netherlands

ABSTRACT

Hardware architecture has long influenced software architecture, and notably so in analytical database systems. Currently, we see a new trend emerging: A "tectonic shift" away from X86-based platforms. Little is (yet) known on how this shift affects database system performance and, consequently, should influence the design choices made. In this paper, we investigate the performance characteristics of X86, POWER, ARM and RISC-V hardware on micro- as well as macro-benchmarks on a variety of analytical database engine designs. Our tool to do so is VOILA: a new database engine generator framework that from a single specification can generate hundreds of different database architecture engines (called "flavors"), among which well-known design points such as vectorized and data-centric execution.

We found that performance on different queries by different flavors varies significantly, with no single best flavor overall, and per query different flavors winning, depending on the hardware. We think this "performance diversity" motivates *a redesign of existing – inflexible – engines towards hardware- and query-adaptive ones.* Additionally, we found that modern ARM platforms can beat X86 in terms of overall performance by up to 2×, provide up to 11.6× lower cost per instance, and up to 4.4× lower cost per query run. This is an early indication that *the best days of X86 are over.*

1 INTRODUCTION

For research as well as industry, query performance is an important topic. To gain an edge (in performance), database engines have to co-evolve with and adapt to the underlying hardware. That meant optimizing for longer CPU pipelines [10, 11], exploiting SIMD [10, 13, 16, 26, 30, 31, 35–37], taking advantage of multiple cores [14, 27] as well as profiting from GPUs, FPGAs or other accelerator devices [19–21, 28, 33, 34]. Besides these (current/past) trends, we can observe a new trend emerging:

After dominating the market for (two - three) decades, mainstream hardware is moving away from X86-based architectures.

The final destination of this shift is yet unclear. Companies like Amazon and Apple push towards ARM architectures (Apple M1 [9], Amazon Graviton [6]), whereas other entities, such as the European Processor Initiative [1] or Alibaba [2], push towards RISC-V-based architectures. ARM-based architectures are in a mature state, they are used in smartphones for years and in several lines of server hardware. RISC-V-based architectures, however, are the somewhat unproven newcomers, with no server-grade hardware being currently available. There are multiple reasons for this shift: at the end of Moore's law, the CISC complexity of X86 may after all these years finally lose out to RISC in raw CPU performance, in a time when X86 manufacturers no longer have a silicon process advantage. Additionally, moving away from X86 offers more freedom and the possibility to add differentiating features: with sufficient funds, one can license an ARM or RISC-V chip design and integrate specific hardware accelerators. The "dark silicon" trend means that performance advances are increasingly going to depend on this.

This shift leaves current, and more so, future database engines, in an uncomfortable spot. Should we continue optimizing for X86? Or should we strive for portable performance, and how would such an engine look like? Besides obvious engineering issues (portability, technical debt, re-implementation effort), there is a chance that performance characteristics will change too. Consequently, this can render existing query execution paradigms, data-centric [29] and vectorized [10], more/less well-suited for a particular hardware, and can invalidate widely known rules of thumb (e.g. vectorized execution outperforms data-centric compilation on data-access-heavy queries). Therefore, the existence of significant performance diversity would require an extensive redesign of current engines.

One possible approach, for such a redesign, is moving to a query execution system that is able to synthesize very different execution strategies (query implementation "flavors"). In our research group, we are building VOILA [17], in which relational operators are implemented in a high-level Domain-Specific Language (DSL), that specifies what operations an operator performs on specific data structures (such as hash tables), eliciting dependencies, but not in what order the operations would be executed. This DSL can then automatically be compiled in different ways into executable implementations (=flavors), for instance using the two opposite approaches of vectorized execution ("x100", after MonetDB/X100 or VectorWise) and data-centric ("hyper"). But, we also integrated ways to mix these two approaches by switching between them at certain points in a query, and added additional variations incorporating SIMD-optimized processing and hardware prefetching. All in all, VOILA can compile the same physical query plan into thousands of different implementation flavors.

A big future research question is how to determine *which* flavor to generate and execute, a question already difficult to answer in the past decade due to the interaction between data distributions, system state and machine characteristics[1]. These interactions can

[1]Given two flavors A and B, A could operate completely in cache, while B might produce cache misses. A could be faster than B (faster memory access). But, for slow

be static (e.g. cache size) as well as dynamic (e.g. processors can clock down to fit into the thermal design budget). While static interactions could be predicted (e.g. using a cost model), dynamic interactions are nearly impossible to predict[2], therefore further complicating the challenge.

Therefore, a solution might lie in *micro-adaptive* [32] execution that adapts the strategy to the circumstances. In VOILA, we eventually aim to dynamically adapt physical implementations (plans, operators, ...) to the currently prevalent (measured) conditions, e.g. via recompilation in a virtual machine-based architecture [15].

But so far, it is unclear, if there is – currently – sufficient performance diversity to justify increasing the flexibility, or even replace existing engines with more flexible ones.

Contributions. We study performance characteristics by conducting micro- as well as macro-benchmarks. We investigate whether there is performance diversity between different platforms and implementations. We study the effects on holistic query performance, hash joins and query execution primitives. We validate the widely-known heuristics: (a) ARM-based platforms are slow and (b) vectorized execution outperforms data-centric compilation on memory-access-heavy workloads. For joins, we identify certain hardware features that tend to favor data-centric compilation.

Structure. The remaining paper is structured as follows: first we discuss the necessary background. Then, we shift the focus to micro-benchmarks that highlight particular aspects of the underlying hardware. Afterwards, we focus on the performance of well-known TPC-H queries, provide insights into the best execution strategy (optimal flavor) on the underlying hardware as well as investigate cost/performance trade-offs. Afterwards, we conclude the paper.

2 BACKGROUND

In this section, we present and discuss the necessary background. We start with a brief description of execution paradigms, then discuss techniques for hiding latencies and, afterwards, related studies.

Data-Centric. Data-centric execution [29] compiles a pipeline into an efficient assembly loop, e.g., lowering the query plan e.g. through an LLVM IR representation. The final code resembles a tuple-at-a-time execution model where each operator is inlined into the next one. Each attribute (of a tuple) is kept in CPU registers (or, if these run out, is spilled to cache memory).

Vectorized Execution. The other well known execution style is vectorized execution [10], which processes data small columnar slices (vectors). The vector size is typically chosen such that execution remains in cache. Data inside a vector is processed in a tight loop. Besides facilitating easy SIMDization via auto-vectorization, this also allows the CPU to speculate ahead.

Simultaneous Multi-Threading (SMT). Hiding the latencies can be a powerful feature. Some modern processors can execute multiple threads on one physical core, a feature called Simultaneous Multi-Threading (SMT). This is effectively increasing instruction-level parallelism, because different threads tend to have mostly independent instruction streams. In theory, this allows hiding memory access or branch misprediction latencies (flushing CPU pipeline only for the affected thread). However, the precise behaviour depends on the hardware at hand, and so does performance.

Prefetching. Besides SMT, it also also possible to hide memory access latencies via prefetching. Starting from the two well-known paradigms (data-centric and vectorized) many variations, with prefetching, have been proposed. Most notably, Asynchronous Memory-Access Chaining (AMAC) [24] and Interleaved Multi-Vectorizing (IMV) [13].

AMAC [24] essentially implements SMT by hand. To benefit from AMAC, one has to implement, e.g. the data-centric pipeline as an array of multiple Finite State Machines (FSMs). The basic idea is, at the end of each state, to prefetch the next data item and switch to a different FSM. Eventually, we reach the original FSM again, with the prefetched data item being available. This allows overlapping prefetching/memory latency with useful work.

IMV [13] improves upon AMAC by introducing efficient buffering. As IMV implements almost all operations via AVX-512, buffering helps keeping SIMD lanes sufficiently filled.

Data-centric vs. Vectorized. The precise advantages and weak-points of data-centric and vectorized execution have been studied by Kersten et al. [22]. They found that vectorization excels in parallel memory access (e.g. hash joins) whereas data-centric shines at computation-heavy workloads. Further, they found that vectorized engine provides other advantages such as accurate profiling, adaptivity and a low compilation time, as primitives are pre-compiled.

3 METHODOLOGY

We conduct micro- to macro-level experiments with varying and, rather, diverse hardware.

Hardware. For our experiments, we used 3 X86 machines (Skylake-X, 8275CL and Epyc), 3 ARM machines (Graviton 1, 2 and M1), 2 PowerPC machines (Power8 and Power9) and one RISC-V machine (910). Specific details can be found in Table 1. Some of the machines have noteworthy special features: The Graviton 2 has accelerated access to always-encrypted memory, as well as acceleration for fast compression and decompression [7]. The M1 features a heterogeneous design of 4 fast CPU cores (Firestorm), 4 slow CPU cores (Icestorm), an integrated GPU and acceleration for Neural Networks. The 910 appears to be an early prototype of a RISC-V-based machine – rather a development board – and appears to target functionality testing, rather than performance (e.g. seems to lack proper cooling). Therefore, we only included the 910 into the basic micro-benchmarks as its performance seems to be the worst of our hardware bouquet. Furthermore, neither Graviton 1, nor M1, nor 910 seem to have an L3 cache.

processors cache miss penalties might matter less, as long as the full main-memory bandwidth can be utilized. Therefore, B could be faster than A.

[2]To predict the full and dynamic behaviour, we would have to assume perfect physical environment (e.g. perfect cooling keeping the system at optimal temperature). Then, we need to model the whole system including hardware quirks. For example the CPU could clock down when using some specific instruction subset (e.g. AVX-512 [12]). For real system, this is (nearly) impossible to "get right" for all supported hardware. It might be possible to build a hardware-specific code model, e.g. by using machine learning to train, or fit, the cost model. The training/fitting step would, however, require extensive data generation, i.e. time-consuming benchmarking, where it is not fully clear yet which features/properties affect performance and need to be measured.

Table 1: Hardware bouquet used in this paper.

	Skylake-X	8275CL	Epyc	Graviton 1	Graviton 2	M1	Power8	Power9	910
Platform	X86	X86	X86	ARMv8.0	ARMv8.2	ARMv8.4	PPC	PPC	RISC-V
Architecture	Skylake-X	Cascade Lake-SP	Zen 2	Cortex-A72	Neoverse N1	Fire-/Icestorm	POWER8	POWER9	C-910
Processor Model	Xeon Gold 6126	Xeon Platinum 8275CL	Epyc 7552	Graviton 1	Graviton 2		POWER8	POWER9	C-910
Threads per Core	1	2	2	1	1	1	8	4	1
Cores per Socket	12	24	48	4	64	4+4	8	16	2
Sockets	2	2	1	4	1	1	2	2	1
NUMA Nodes	2	2	1	1	1 [7]	1	2	2	1
RAM (GiB)	192	192	192	32	128 [3]	16	256	128	4
L3 (MiB)	19.25	35.75	192	-	32 [3]	-	64	120	-
L2 (kiB)	1024	1024	512	2048 [4]	1024 [3]	4096/2048	512	512	2048
L1d (kiB)	32	32	32	32 [4]	64 [3]	128/64	64	32	64
L1i (kiB)	32	32	32	48 [4]	64 [5]	192/128	32	32	64
Freq. max (Ghz)	3.7	3.6	3.5	2.3 [4]	2.5 [5]	3.2/2	3.0	3.8	1.2
Freq. min (Ghz)	1.0	1.2	1.8	2.3 [4]	2.5 [5]	0.6/0.6	2.0	2.1	1.2
AWS Instance	-	c5.metal	c5a.24xlarge	a1.metal	r6gd.metal	-	-	-	-

Synthesizing Efficient Implementations. Rather than implementing the required queries by hand, we synthesize them. VOILA [17] allows synthesizing many implementations from one query description. It generates data-centric and vectorized flavors that perform on-par with handwritten implementations and, implicitly, the systems Hyper [29] and Vectorwise [10]. VOILA also allows generating mixes that facilitate prefetching and efficient overlapping of prefetching with useful computation, similar to AMAC [24] and IMV [13]. In this paper, we use the VOILA-based synthesis framework to generate the implementations required for our experiments on holistic query performance. To investigate such impacts, we ported the VOILA [17] synthesis framework to non-X86 architectures.

4 MICRO-BENCHMARKS

We start with micro-benchmarks that stress specific aspects of the underlying hardware: (a) memory access, (b) data-parallel computation and (c) control flow & data dependencies.

For each type, we implemented vectorized primitives, functions that operate on columnar vectors in a tight loop. We ran the micro-benchmarks multi-threaded using all available threads to the operating system (i.e. including SMT, if available). We report per-tuple timings in nanoseconds, normalized by the number of SMT threads (e.g. for 1 real core and 8 SMT threads, we divide the time by 8).

4.1 Memory Access

In modern database engines, memory access is a well-known bottleneck for certain queries [11, 18]. Therefore, we investigate memory access performance. We differentiate between (1) cache-mostly random reads, (2) bigger-than-cache random reads and (3) the `SUM` aggregate as a read-update-write workload. Each experiment accesses an array of 64-bit integers at pseudo-random indices.

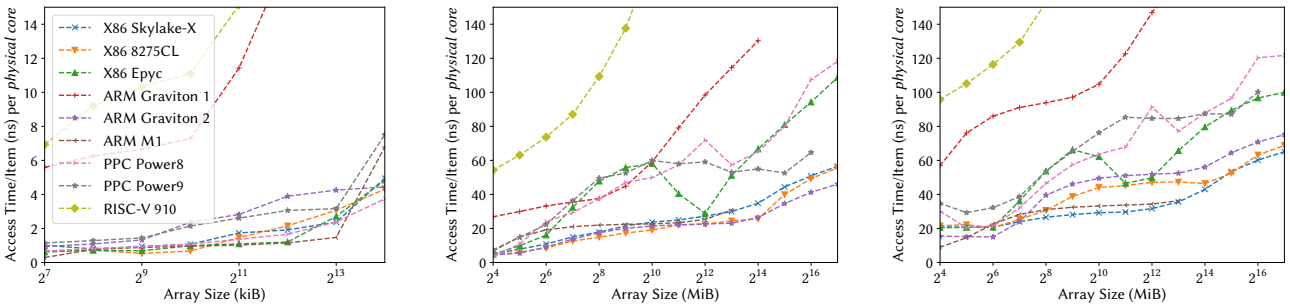
Cache-mostly random Reads. The runtimes for each random access read from a small array are plotted in Fig. 1a. In general, we see very little difference, but two extreme outlier: The Graviton 1 and the 910 feature very slow cache access, of which the 910 shows the worst performance. Faster than Graviton 1, but slower than the others (Skylake-X, 8275CL, Eypc, Power8) are Power9 and Graviton 2. Both provide relatively slow access to small-to-medium-small arrays. The Graviton 2, however, can "catch up with the crowd", for slightly bigger arrays

Main-Memory-mostly random Reads. We now move to large arrays which likely do not fully reside in caches. That means that on NUMA machines, we might see NUMA overheads (data needs to be shipped from one socket to another). As very large arrays do not fit into one NUMA node, we decided to interleave the whole array over all NUMA nodes (round-robin assignment of pages to NUMA nodes, page `i` assigned to node `i % num_nodes`). Essentially, this resembles the initial bucket lookup of the scalable non-partitioned hash join described by Leis et al. [27] which also effectively interleaves the array over all NUMA nodes.

For large arrays, the random reads are visualized in Fig. 1b. We see a rather big divergence, especially with growing array size.

Skylake-X, 8275CL and Graviton 2 perform best, of which the Graviton 2 outperforms the other two for arrays $> 2^{14}$ integers. It has to be said that Graviton 2 offers direct non-NUMA memory access whereas, both, the Skylake-X and the 8275CL, are NUMA machines and might require transferring data from other sockets over a shared bus. While this is a rather uncommon architectural decision, this gives the Graviton 2 an advantage as it can avoid costly data transfers over a shared bus and use fast local memory instead. The other competitive non-NUMA machine is the Epyc but it tends to feature rather slow memory access for L3-mostly (until 2^7 MiB) and main-memory-mostly random reads.

`SUM` *Aggregate.* The `SUM` is a read-update-write workload and will, therefore, for many core systems involve a cache invalidation cost (invalidate, write back and possibly read from another core or socket). However, due to the relatively large size of the array, the chance of such false sharing (writing a cache line that is, later, read and modified by another core or socket) is relatively low. Figure 1c shows our measurements. The measurements look similar to random-read workload, but tend to be slower. We can observe that the Skylake-X and 8275CL outperform for large arrays, followed by the Graviton 2, which outperforms for arrays $\leq 2^7$ MiB.



(a) Cache-mostly random Reads **(b) Main-memory-mostly random Reads** **(c) Main-memory-mostly random `SUM`**

Figure 1: Memory-heavy workloads.

Conclusions. Both, the Graviton 1 and the 910, show slow memory access across the board. Power8 and Epyc are slower for larger arrays. The Skylake-X and 8275CL are fast across the board. Graviton 2 is relatively slow (compared to Skylake-X, 8275CL) on smaller arrays ($\leq 2^{13}$ kiB), but outperforms for large arrays ($\geq 2^{15}$ MiB).

4.2 Data-Parallel Computation

Besides memory access, the other important cornerstone for query performance is computational power. As examples of computation-heavy workloads, we performed a series of additions and multiplications in a tight loop. We investigate the performance of relatively cheap (addition) vs. relatively pricey (multiplication) operations, and the impact of thin data types [16]. For vectorized kernels, there are two alternative paths that influence performance: (a) the non-selective path that only fetches data from input vectors, computes the result and writes the output, and (b) the selective path which introduces an additional indirection.

The non-selective path accesses plain arrays (vectors) in sequential order and is, thus, amenable to acceleration with SIMD. Thanks to the trivial access pattern, the compiler will typically automatically SIMDize this code path.

Selective execution intends to only process selected tuples (i.e. efficiently ignore values that are filtered out). We implemented this path, like described by the classic vectorized execution model [10], using selection vectors that describe which indices in the vectors are alive. Due to this indirection, the compiler will typically not SIMDize this path. To reduce branch prediction overhead (in the `for` loop), we unrolled this path 8 times.

Performance. Our results can be found in Fig. 2. At the first glance, we see that selective computations (transparent) are significantly more expensive than non-selective computations (opaque). This is due to the extra indirection which is (a) expensive and (b) prevents efficient SIMDization. To alleviate this overhead, in a vectorized system one would, typically, ignore the selection vector, when the vector more than, say, 30% full ("full computation" [32]).

Between machines, we also see significant differences. The 910 shows "off the charts" performance for 32-bit additions and multiplications and is roughly 24× slower than the Skylake-X or the Epyc. The Graviton 1 significantly faster than the 910, but is roughly 4× slower than the Skylake-X or the Epyc. The fastest machines are Skylake-X, 8275CL and Epyc which are roughly 2× faster than the

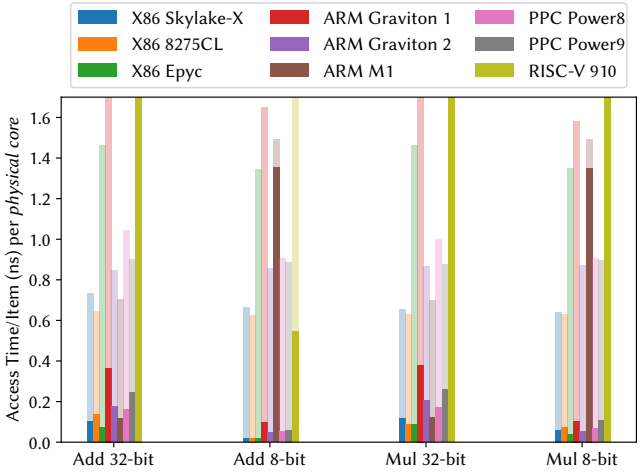
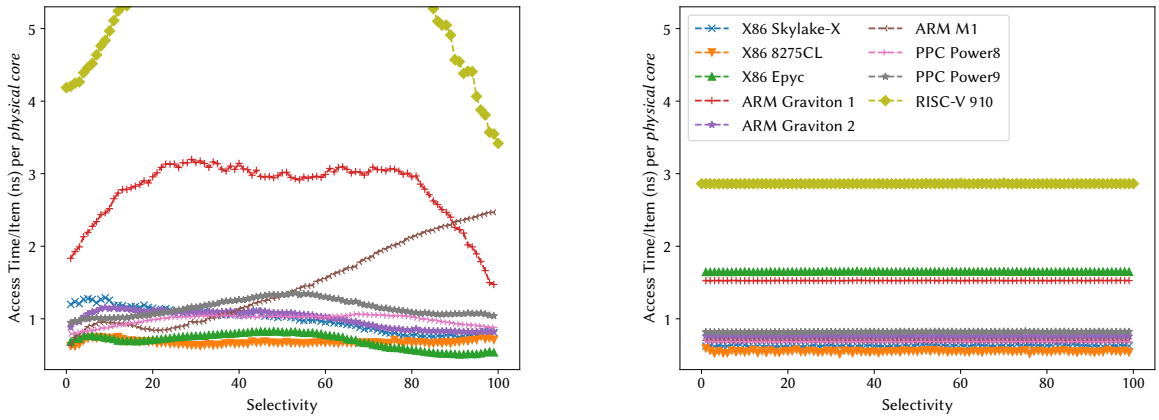


Figure 2: Computation without selection vector, amendable to SIMD acceleration (opaque). Computation with selection vector introduces extra indirection (transparent).

other machines. For ARM platforms, the slowdown is likely caused by the lack of compiler auto-vectorization. For non-selective workloads, thin data types provide significant benefit on most platforms. However, this benefit disappears when the selection vector is used. Notable exception is the M1 which not only lacks vectorization benefits but also incurs additional overhead for loading and storing 8-bit integers. For the 8-bit addition the M1 even performs worse than the 910.

Scalability. Typically, when using all available cores, processors tend to significantly reduce computational throughput. This is typically due to heat emission. Each chip has a given thermal budget (thermal design power, TDP): If the budget is reached, heat emission needs to be curtailed. Therefore, cores clock down and, thus, scale computational throughput down.

In this experiment, we run many SIMDizable (non-selective) 32-bit integer multiplications with varying degrees of parallelism (DOP, or number of threads N). We scale the DOP from 1 (no parallelism) to the level parallelism the hardware provides (T). Note that this experiment evaluates the *best-case*, as the scalability of



(a) Branch-based creation **(b) Dependency-based creation**

Figure 3: Control-Flow & Data-Dependency-intensive workload: Creating a selection vector from an 8-bit boolean predicate.

Table 2: Graviton 1 & 2 provide consistent performance over multiple degrees of parallelism (DOP, N), while many others show significant slowdowns. (32-bit integer multiplication without selection vector, without adjustment for SMT, DOP beyond real cores are marked in italics on gray)

Hardware		Time/Item in ns (slowdown) on given DOP N				
	T	N=1	$T/8$	$T/4$	$T/2$	T
X86 Skylake-X	24	0.08	0.08 (1.0×)	0.09 (1.1×)	0.09 (1.2×)	0.12 (1.5×)
X86 8275CL	96	0.08	0.08 (1.0×)	0.10 (1.4×)	0.17 (2.3×)	*0.18 (2.3×)*
X86 Epyc	96	0.08	0.08 (1.0×)	0.08 (1.0×)	0.12 (1.5×)	*0.17 (2.3×)*
ARM Graviton 1	16	0.34	0.34 (1.0×)	0.36 (1.1×)	0.35 (1.0×)	0.38 (1.1×)
ARM Graviton 2	64	0.20	0.20 (1.0×)	0.20 (1.0×)	0.20 (1.0×)	0.20 (1.0×)
ARM M1	8	0.07	*is $N = 1$*	0.08 (1.1×)	0.08 (1.1×)	0.12 (1.7×)
PPC Power8	128	0.26	0.38 (1.4×)	*0.68 (2.6×)*	*1.27 (4.8×)*	*1.38 (5.2×)*
PPC Power9	128	0.19	0.27 (1.4×)	0.52 (2.7×)	*0.95 (5.0×)*	*1.04 (5.4×)*

```
int select_true(int* res, i8* a, int* sel, int n) {
  int r = 0;
  if (sel) {
    for (int i=0; i<n; i++) {
      if (a[sel[i]]) res[r++] = sel[i];
    }
  } else {
    for (int i=0; i<n; i++) {
      if (a[i]) res[r++] = i;
    }
  }
  return r;
}
```

Listing 1: Vectorized kernel: Creating a selection vector using branches

OLAP queries is typically limited by other factors, such as memory-access. Table 2 shows our results.

With increasing DOP, we see a tendency to significant slowdowns of 50% to 2.7× without using SMT cores and up to roughly 5× with SMT cores. However, with the exception of the Graviton-based platforms which, evidently, do not clock down significantly. M1 showed less throughput beyond 4 cores. This can be explained by the design of the M1 that combines 4 fast and 4 slow cores, i.e. workloads with >4 threads ($> T/2$) will also use the slower cores.

4.3 Control Flow & Data Dependencies

Besides data-parallel computation, modern engines typically also rely on fast control flow and data dependencies. Depending on the hardware (e.g. pipeline length), branch misses can become quite costly, and data dependencies reduce the CPU pipeline parallelism.

In vectorized engines, such operations appear when creating a selection vector (e.g. in a filter or hash-based operators). Therefore, we benchmark the performance of selection vector creation.

Selection vectors can be built in multiple ways: Most commonly, they are created using (a) branches or (b) data dependencies. Alternatively, one can create selection vectors using using X86-specific AVX-512 `vpcompressstore` [22]. While this method is often faster [22], creating selection vectors using AVX-512 is not portable to other hardware architectures.

A branch-based implementation as in Listing 1, stresses the branch predictor. For very high/low selectivities, the branch becomes predictable. The closer the selectivity comes to 50%, the more unpredictable (or harder to predict) the branch becomes.

As an alternative to creating selection vectors using branches, one can introduce a data dependency. In pseudo-code, in Listing 1, this means replacing `if (a[k]) res[r++] = k` by `res[r] = k; r+= a[k]`. Obviously, this avoids the overhead of mispredicting branches, but might introduce additional costs for predictable branches.

Results. For the branch-based creation of selection vectors, our results can be found in Fig. 3a. We refer to selectivity as the fraction of tuples that pass the filter (100% = all pass). Typically, one would expect lower timings (faster) for very low and high selectivities, because the branch becomes predicable. Around 50% selectivity, one would expect the worst performance as the branch is unpredictable. We can observe this behaviour on the Epyc and Power9. The plot of the Graviton 1 stands out due to very high cost, and edge behavior at middle-low and -high selectivities, which are more expensive than branches of 50%. Also interesting, is the plot of the M1, which exhibits an asymmetrical shape where taking the branch (`if(a[i])`) is more expensive than skipping it.

Table 3: Best execution paradigm unclear for simple join query. Best flavor as tuple (Computation Type, Prefetch, #FSMs), data-centric in blue and prefetching in italics.

	Best Flavor		Well-known Flavors	
	Name	best (ms)	x100 (ms)	hyper (ms)
X86 Skylake-X	*vec(1024),3,1*	173	209	239
X86 8275CL	*vec(512),3,2*	176	192	236
X86 Epyc	*scalar,4,16*	134	139	157
ARM Graviton 1	scalar,0,1	412	440	412
ARM Graviton 2	vec(1024),0,1	101	101	115
ARM M1	*vec(1024),4,1*	228	273	297
PPC Power8	*vec(512),2,1*	488	498	507
PPC Power9	*scalar,3,1*	317	339	317

We now juxtapose the performance of branch-based (Fig. 3a) with the dependency-based selection vector creation (Fig. 3b). In general, dependency-based creation tends to outperform, with one notable exception: The Epyc for which the branch-based created is faster. For the 8275CL, both variants are roughly equal. Thus, it can be said that the best way to create selection vector depends on the hardware at hand. In a system, it is advisable to determine the choice between branch- and dependency-based selection vector building adaptively at runtime.

4.4 Case Study: Hash Join

As we gradually move towards macro-benchmarks, we now investigate the performance of a hash join. In particular, we are interested whether the heuristic *vectorized execution excels in data-access-heavy workloads* is true on different hardware/platforms.

Therefore, we synthesized the following SQL query using the VOILA-based synthesis framework [17]:

```
SELECT count(*) FROM lineitem, orders
WHERE o_orderdate < date '1996-01-01'
  AND l_quantity < 50 AND l_orderkey = o_orderkey
```

We ran this query in multiple flavors on the TPC-H data set with scale factor 10. Table 3 shows the best flavor as well as the runtimes of the data-centric (*hyper*) and vectorized (*x100*) execution.[3]

We see that the majority of the best flavors are indeed vectorized. To our surprise, data-centric flavors can beat vectorized flavors. Typically, the winning data-centric flavors need elaborate prefetching to outperform, with one notable exception: On the Graviton 1, the plain data-centric flavor (without Finite State Machines and without prefetching) outperforms the vectorized flavors.

Favorable Features for Data-Centric Execution. We believe that on the Graviton 1 the $3-4\times$ slower computation (*slower cores*, Fig. 2) favors data-centric execution, because the more efficient computation (data-centric, less issued instructions) outweighs the less efficient memory-access.

The other machines feature faster cores. The *very large L3 caches*, on the Epyc and Power9, tend to benefit data-centric flavors, as huge L3 caches leads to effectively faster memory access (more data in faster memory) and, thus, making more efficient memory access (using vectorized execution) less important. This is further exaggerated via *SMT* which can effectively hide memory access latency by executing another thread. On the Epyc, both features (SMT and large L3) are barely enough to allow a data-centric flavor to win (134ms vs. 139ms, roughly on noise level). We notice similar behaviour on the Power8/Power9, which feature a large L3 cache and, compared to the Epyc, a higher degree of SMT (8 threads on Power8, or 4 threads on Power9, vs. 2 threads per core).

In summary, we can say that, certain hardware properties (slow cores, large L3 cache and SMT) have a tendency to favor data-centric execution, for joins.

[3]We use x100 as short identifier for vectorized execution, in reference to MonetDB/X100 [10] – which later became Vectorwise and currently is called Vector.

Table 4: Graviton 2 beats all other machines in overall performance. M1 leads in performance per core. Runtimes of well-known query execution paradigms.

	Q1		Q3		Q6		Q9	
	x100	hyper	x100	hyper	x100	hyper	x100	hyper
Runtime (milliseconds)								
X86 Skylake-X	79	54	261	282	28	35	228	291
X86 8275CL	93	84	480	397	70	112	232	261
X86 Epyc	81	65	241	238	51	52	193	180
ARM Graviton 1	188	107	447	447	55	45	720	715
ARM Graviton 2	**42**	**29**	**162**	**158**	**20**	**22**	**95**	**109**
ARM M1	216	86	313	440	138	404	432	590
PPC Power8	404	384	1094	1132	336	337	627	636
PPC Power9	239	225	645	631	190	192	406	393
*Runtime * Number of real cores (seconds)*								
X86 Skylake-X	1.9	1.3	6.3	6.8	**0.7**	0.8	5.5	7.0
X86 8275CL	4.5	4.1	23.1	19.0	3.4	5.4	11.1	12.5
X86 Epyc	3.9	3.1	11.6	11.4	2.4	2.5	9.3	8.6
ARM Graviton 1	3.0	1.7	7.1	7.2	0.9	**0.7**	11.5	11.4
ARM Graviton 2	2.7	1.9	10.4	10.1	1.3	1.4	6.1	7.0
ARM M1	**1.7**	**0.7**	**2.5**	**3.5**	1.1	3.2	**3.5**	**4.7**
PPC Power8	6.5	6.1	17.5	18.1	5.4	5.4	10.0	10.2
PPC Power9	7.6	7.2	20.6	20.2	6.1	6.1	13.0	12.6

5 MACRO-BENCHMARKS

While micro-benchmarks provide useful insights into extreme cases, it is hard to draw conclusions on holistic query performance. Queries are more complex than simple operations and are, thus, rarely completely limited by either memory bandwidth, or computational throughput. Using VOILA, we generated implementations for TPC-H Q1, Q3, Q6 and Q9. For our benchmarks, we used the TPC-H data set with scale factor 10. For each query, we sampled 50 different execution flavors from the universe that VOILA can generate, always including the two most well-known ones: pure data-centric compilation [29] and pure vectorized execution [10].

5.1 Query Performance

Here, we compare the runtime of data-centric (*hyper*) and vectorized (*x100*) on varying hardware in terms of overall system performance and per-core performance. The results are visualized in Table 4.

Overall Performance. We notice a significant diversity in overall runtimes of up to, roughly, $10\times$ between the fastest and the slowest machine. Common wisdom would suggest that X86 would perform best, but surprisingly the ARM Graviton 2 significantly outperforms all others. Compared to the runner-up (Skylake-X), it performs up to $3\times$ faster (Q9 hyper) and, often executes queries roughly $2\times$ faster (Q1, Q3, Q9).

Table 5: Best query execution paradigm unclear, even for specific queries (e.g. Q9). Best flavor as tuple (Computation Type, Prefetch, #FSMs). Data-centric (scalar) flavors are marked in blue color, prefetching in italics.

	Q1	Q3	Q6	Q9
X86 Skylake-X	scalar,0,1	*vec(2048),3,1*	*vec(1024),3,1*	*vec(1024),2,1*
X86 8275CL	*scalar,2,1*	*scalar,3,8*	*vec(512),4,1*	*scalar,2,32*
X86 Epyc	*scalar,2,1*	*vec(256),1,1*	*vec(1024),2,1*	vec(512),0,1
ARM Graviton 1	scalar,0,1	vec(512),0,1	*scalar,4,1*	vec(256),0,1
ARM Graviton 2	*scalar,2,1*	scalar,0,1	*vec(2048),2,1*	vec(512),0,1
ARM M1	scalar,0,1	*vec(2048),2,1*	*scalar,3,1*	*vec(1024),2,1*
PPC Power8	scalar,0,1	*vec(1024),2,1*	vec(256),0,1	*scalar,2,2*
PPC Power9	*scalar,3,1*	vec(512),0,1	*vec(256),2,1*	*scalar,2,8*

On Q9, we see data-centric flavors outperforming vectorized flavors, for the Epyc, Graviton 1 and Power9. This is due to the hardware properties we identified in Section 4.4: very large L3 caches (Epyc, Power9), SMT (Epyc, Power9) as well as slow cores (Graviton 1). These factors favor data-centric execution on join-intensive workloads such as Q9 in particular.

Another important query is Q3, which is less join-heavy and is, therefore, less suited to vectorized execution. We noticed that data-centric outperforms on 8275CL, Epyc, Graviton 2, Power8 and Power9. This is partially caused by the hardware factors we identified (L3 size, SMT, slow cores) and, partly, by the structure of the query.

Per-Core Performance. We scale the multi-threaded performance up to the number of real cores. This provides a per-core performance metric that includes potential multicore scalability bottlenecks (e.g. down clocking, memory bandwidth). Implicitly, this metric favors systems with a lower number of cores (typically close to desktop systems).

The machines with a lower number of cores (Graviton 1, M1 and Skylake-X) score best. This is partly an artifact of queries not scaling perfectly sometimes, but it gives an indication of real per-core performance (i.e., in a parallel workload). This per-core performance is dominated by ARM platforms, except for *Q6 x100*. But to our surprise, the Graviton 2, *with 64 cores*, comes close to the best 3. This indicates that Graviton 2 scales quite well, up to all 64 cores, on whole queries and not just in micro-benchmarks (Section 4.1 and Section 4.2).

5.2 Optimal Flavor

Here, we investigate which execution paradigm (flavor) is the best for each query. Using the VOILA-based synthesis framework, we generated basic flavors i.e. one flavor per query (no mixes). Table 5 shows the flavors with the lowest average runtime.

Best Flavor. We can see that some configurations, most notably Skylake-X and Epyc, exhibit the behaviour described by Kersten et al. [22] that data-centric wins in Q1 and vectorized wins in Q3 and Q9. However, surprisingly, we found data-centric flavors winning on the join-heavy Q3 and Q9. Notably, these are augmented data-centric flavors with prefetching and multiple finite state-machines (FSMs) that allow overlapping prefetching with useful computation. In particular, these augmented data-centric flavors perform well on large machines with multiple threads per core (SMT), i.e. 8275CL, Power8 and Power9.

Table 6: Best flavors outperform by up to 220%. Runtime improvement over plain vectorized (*x100*)/data-centric (*hyper*).

	Best vs. *x100* (%)				Best vs. *hyper* (%)			
	Q1	Q3	Q6	Q9	Q1	Q3	Q6	Q9
X86 Skylake-X	47	3	5	13	*is*	12	29	44
X86 8275CL	14	54	12	16	4	27	79	31
X86 Epyc	29	8	3	19	3	6	7	11
ARM Graviton 1	77	1	22	12	*is*	1	0	11
ARM Graviton 2	62	3	15	2	12	*is*	23	17
ARM M1	150	7	10	9	*is*	51	220	49
PPC Power8	5	9	3	9	*is*	12	3	11
PPC Power9	7	3	3	10	1	1	5	7

Although, the winning flavor on Q1 is data-centric (scalar), we can see the use of prefetching for a query that runs mostly in cache. This is caused by the low overhead introduced by prefetching rather than the actual benefit of prefetching (improvements between data-centric and best flavor in Q1 are on noise level i.e. < 15%).

Are the "best Flavors" really better? Table 6 shows the improvement of the best flavors over the well-known data-centric (*hyper*) and vectorized (*x100*). In many cases, the best flavor outperformed the well-known ones by 10-31%. In some cases the difference was on noise level, but other cases the best flavor significantly outperforms well-known ones by up to 220%. Therefore, we can conclude that there is significant performance diversity, which – in some cases – can be exploited. However, taking advantage of this diversity, in practice, would require significantly more flexible engines.

5.3 Costs & "Bang for the Buck"

While the ARM Graviton 2 might outperform the other machines on performance, it may not necessarily provide the best performance on a price-adjusted basis. Therefore, we investigated the costs for renting the hardware, used for our experiments, and discuss cost performance trade-offs.

For pricing, we used the spot prices reported on AWS [8]. Unfortunately, PowerPC architectures and the M1 were not available. Even though we did not use AWS for our Skylake-X machine, we found a similar instance type (*z1d.12xlarge*) that we used for pricing.

Cost. We visualized the costs in Table 7. From that table it is evident that ARM-based instances are up to 12× cheaper per hour and 11× cheaper per core. The most expensive instance is the Skylake-X. It is also the best performing X86 machine (Q1, Q3, Q6 and Q9) and is only beaten on Q9 by the Epyc.

Cost per Q9 run. Typically, faster machines are more expensive. Therefore, we calculated the cost for 1 million runs of vectorized flavor of TPC-H Q9 with scale factor 10.

On this metric, the ARM instances outperform by > 2×. Compared to the cheapest X86-instance (Epyc), the Graviton 1 is 3× cheaper per run whereas the Graviton 2 is 2.5× cheaper.

6 CONCLUSIONS & FUTURE WORK

Performance diversity has become ubiquitous as hardware is getting more heterogeneous; challenging system architects in the design choices they make. For instance, Graviton 2 is comparatively slow on smaller arrays while being better on large arrays, thin data types [16] are heavily penalized on the M1 and there is no clear

Table 7: ARM Gravitons are significantly cheaper and provide most "bang for the buck".

	$\frac{\$}{\text{hour}}$	$\frac{\text{Cents per real core}}{\text{hour}}$	Q9 (ms)	1M × Q9 ($)
X86 Skylake-X (price est.)	1.3392	5.6	228	84
X86 8275CL	0.9122	1.9	232	59
X86 Epyc	0.9122	1.9	193	49
ARM Graviton 1	**0.0788**	**0.5**	720	**16**
ARM Graviton 2	0.7024	1.1	**95**	19

portable optimum for building selection vectors. Neither vectorization nor data-centric compilation are optimal in all cases, even in data-access-heavy workloads, like hash joins or TPC-H Q9. The heuristic *vectorization outperforms data-centric in data-access-heavy workloads* is not generally true, as on some platforms data-centric flavors outperform. Prefetching can be useful to further boost the performance of data-centric flavors.

We identified several hardware features that, for joins, tend to favor data-centric execution: Very large L3 caches, Simultaneous Multi-Threading (SMT) and slow cores. On one hand, we have features that improve data-centric execution's less efficient memory access by fitting more data in cache (very large L3) or effectively hiding memory access latencies (SMT). On the other hand, some machines only feature very slow cores (most notably the ARM Graviton 1) which penalizes more efficient memory access (i.e. vectorized execution) and, hence, "making data-centric look better".

We also confirm that modern ARM platforms are now outclassing X86 on these analytical database workloads. The ARM Graviton 2 is now up to 2× faster than the fastest X86 machine. Single-core performance is better (ARM M1) or on par with X86 (Graviton 2). When using all cores, the Gravitons provide constant computational throughput, while X86 machines typically limit the throughput (to stay within thermal budget). In the cloud, ARM architectures are significantly cheaper (up to 11.6×) and provide up to 4.4× higher performance per dollar. Modern ARM platforms are not only faster (Graviton 2) but also cheaper (Graviton 1 and 2) to a larger degree.

Consequences & Future Work. We have seen that (a) modern ARM platforms can outperform X86, (b) the best execution flavors are hardware-dependent and (c) best flavors can significantly improve performance (by up to 220%). Consequently, our conclusion is that database architecture needs to move to more flexible and adaptive engines, not solely focused on X86 (like e.g. Umbra [23]) or even one specific execution paradigm.

As a way forward on this path, we propose to investigate the introduction of *database virtual machines* into query engines, that automatically generate flavors and adaptively discover the best one for the current hardware and workload. Such a virtual machine could use a domain-specific language (as used by our VOILA framework [17]) to synthesize the best paradigm, either on a coarse-grained pipeline level [25], or fine-grained per code fragment [15].

REFERENCES

[1] 2021. https://www.european-processor-initiative.eu/. Accessed: 2021-03-02.
[2] 2021. https://www.nextplatform.com/2020/08/21/alibaba-on-the-bleeding-edge-of-risc-v-with-xt910/. Accessed: 2021-03-02.
[3] 2021. https://www.forbes.com/sites/moorinsights/2019/12/03/aws-goes-all-in-on-arm-based-graviton2-processors-with-ec2-6th-gen-instances/. Accessed: 2021-03-02.
[4] 2021. https://en.wikichip.org/wiki/annapurna_labs/alpine/al73400. Accessed: 2021-03-02.
[5] 2021. https://www.anandtech.com/show/15578/cloud-clash-amazon-graviton2-arm-against-intel-and-amd. Accessed: 2021-03-02.
[6] Amazon. 2021. https://aws.amazon.com/en/ec2/graviton/. Accessed: 2021-03-02.
[7] Amazon. 2021. https://d1.awsstatic.com/events/reinvent/2019/REPEAT_1_Deep_dive_on_Arm-based_EC2_instances_powered_by_AWS_Graviton_CMP322-R1.pdf. Accessed: 2021-03-02.
[8] Amazon. 2021. https://aws.amazon.com/de/ec2/spot/pricing/. Accessed: 2021-03-19.
[9] Apple. 2021. https://www.apple.com/mac/m1/. Accessed: 2021-03-02.
[10] Peter Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution.. In *CIDR*. 225–237.
[11] Peter A Boncz, Martin L Kersten, and Stefan Manegold. 2008. Breaking the memory wall in MonetDB. *Commun. ACM* 51, 12 (2008), 77–85.
[12] Travis Downs. 2020. https://travisdowns.github.io/blog/2020/08/19/icl-avx512-freq.html. Accessed: 2021-06-21.
[13] Zhuhe Fang, Beilei Zheng, and Chuliang Weng. 2019. Interleaved Multi-vectorizing. *PVLDB* 13, 3 (2019), 226–238.
[14] Goetz Graefe. 1990. Encapsulation of Parallelism in the Volcano Query Processing System. In *SIGMOD*. 102–111.
[15] Tim Gubner. 2018. Designing an adaptive VM that combines vectorized and JIT execution on heterogeneous hardware. In *Proc. ICDE*.
[16] Tim Gubner and Peter Boncz. 2017. Exploring Query Execution Strategies for JIT, Vectorization and SIMD. In *ADMS@VLDB*.
[17] Tim Gubner and Peter Boncz. 2021. Charting the Design Space of Query Execution using VOILA. In *PVLDB*, Vol. 14. 1067–1079.
[18] Tim Gubner, Viktor Leis, and Peter Boncz. 2020. Efficient Query Processing with Optimistically Compressed Hash Tables & Strings in the USSR. In *ICDE*.
[19] Tim Gubner, Diego Tomé, Harald Lang, and Peter Boncz. 2019. Fluid co-processing: GPU bloom-filters for CPU joins. In *DaMoN@SIGMOD*. 1–10.
[20] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *PVLDB* 6, 10 (2013), 889–900.
[21] Tomas Karnagel, Matthias Hille, Mario Ludwig, Dirk Habich, Wolfgang Lehner, Max Heimel, and Volker Markl. 2014. Demonstrating Efficient Query Processing in Heterogeneous Environments. In *SIGMOD*. 693–696.
[22] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *PVLDB* (2018), 2209–2222.
[23] Timo Kersten, Viktor Leis, and Thomas Neumann. 2021. Tidy Tuples and Flying Start: Fast Compilation and Fast Execution of Relational Queries in Umbra. *VLDB Journal* 30 (2021).
[24] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous memory access chaining. *PVLDB* 9, 4 (2015), 252–263.
[25] André Kohn, Viktor Leis, and Thomas Neumann. 2018. Adaptive execution of compiled queries. In *Proc. ICDE*. IEEE, 197–208.
[26] Harald Lang, Linnea Passing, Andreas Kipf, Peter Boncz, Thomas Neumann, and Alfons Kemper. 2020. Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines. *VLDB Journal* 29, 2 (2020), 757–774.
[27] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age. In *SIGMOD*. 743–754.
[28] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *PVLDB* 2, 1 (2009), 229–240.
[29] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *PVLDB* 4, 9 (2011), 539–550.
[30] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *SIGMOD*. 1493–1508.
[31] Orestis Polychroniou and Kenneth A Ross. 2014. Vectorized bloom filters for advanced SIMD processors. In *DaMoN@SIGMOD*. 1–6.
[32] Bogdan Răducanu, Peter Boncz, and Marcin Zukowski. 2013. Micro adaptivity in vectorwise. In *SIGMOD*. 1231–1242.
[33] David Sidler, Zsolt István, Muhsen Owaida, and Gustavo Alonso. 2017. Accelerating pattern matching queries in hybrid CPU-FPGA architectures. In *SIGMOD*. 403–415.
[34] David Sidler, Zsolt István, Muhsen Owaida, Kaan Kara, and Gustavo Alonso. 2017. doppiodb: A hardware accelerated database. In *SIGMOD*. 1659–1662.
[35] Evangelia Sitaridi et al. 2016. SIMD-accelerated Regular Expression Matching. In *DaMoN@SIGMOD*. 8:1–8:7.
[36] Thomas Willhalm, Nicolae Popovici, Yazan Boshmaf, Hasso Plattner, Alexander Zeier, and Jan Schaffner. 2009. SIMD-scan: Ultra Fast In-memory Table Scan Using On-chip Vector Processing Units. *PVLDB* 2, 1 (2009), 385–394.
[37] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *SIGMOD*. 145–156.