Einreichung für: BTW 2017,
Geplant als Veröffentlichung innerhalb der Lecture Notes in Informatics (LNI)

Hardware-Sensitive Scan Operator Variants for Compiled Selection Pipelines

David Broneske[1], Andreas Meister[1], Gunter Saake[1]

Abstract: The ever-increasing demand for performance on huge data sets forces database systems to tweak the last bit of performance out of their operators. Especially query compiled plans allow for several tuning opportunities that can be applied depending on the query plan and the underlying data. Apart from classical query optimization opportunities, it includes to tune the code using code optimizations for processor specifics, e.g., using Single Instruction Multiple Data processing or predication. In this paper, we examine code optimizations that can be applied for compiled scan pipelines that include aggregations, evaluate impact factors that influence the performance of the scan pipelines, and derive guidelines that a query compiler should implement to choose the best variant for a given query plan and workload.

Keywords: Multi-Column Selection Predicates, Scans, Hardware Sensitivity, SIMD, Predication

1 Introduction

With the advent of main-memory databases, where all necessary data to be processed fits into memory, good performance of database operator code has become a key challenge [Ba13; Br14]. A superior method to produce fast, optimized code for database queries is query compilation [Ne11]. In essence, a modern query compiler will split a query into several[3] so-called *pipelines* that merge the code of a set of operators into a single loop.

Important ingredients for a pipeline are `if`-clauses, that represent selections. Inside the `if`-clauses, there are further operators, such as projections and hash-table probes and, at last inside the `if`-clause, code for pipeline breakers, such as aggregations or hash-table builds. Having selections in the pipeline, the code becomes prone to branch mispredictions due to the if-clause. In an earlier work, we already experimented with several optimizations of a selection operator [BBS14]. Among others, we evaluated the performance of a branching scan against a branch-free (predicated) variant of the scan and also against a scan using the concept of *Single Instruction Multiple Data* (SIMD). The result was, as shown in Figure 1.a, that the predicated variant is overall the best choice. However, these experiments are limited to single predicate selections materialized into a position list; whereas, query compilation allows for several predicates merged into a pipeline including further code inside the loop, for instance, for executing the aggregation. Both scenarios can be found in the TPC-H queries [Tr14], e.g., in query `Q1` and query `Q6`. `Q1` consists of a selection

[1] University of Magdeburg, Database and Software Engineering Group, Universitätsplatz 2, 39106 Magdeburg, firstname.lastname@ovgu.de. This work was partially funded by the DFG (grant no.: SA 465/50-1).

[3] A query may have to be split into several of these pipelines, whenever there is a pipeline breaker (e.g., a join) that needs to consume all the input tuples for the query to proceed.

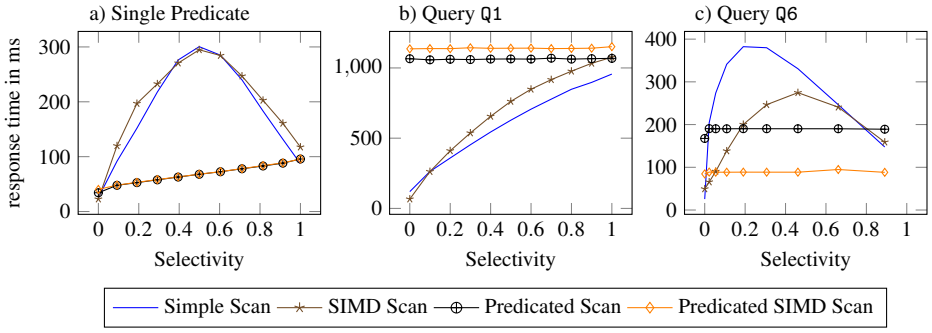

Fig. 1: Response time for different scan implementations on a TPC-H scale factor 10 Lineitem table.

on a single predicate with six aggregates inside the loop, while Q6 has three predicates and a single aggregate. In Figure 1, we depict the runtime of both queries for varying selectivities. In contrast to the single predicate, where the predicated variants are superior, for query Q1 the branching variants are superior, because the overhead of predication that is added to the work inside the loop is too heavy. Furthermore for query Q6, we can see that a SIMD scan is beneficial compared to the normal scan for a range of selectivities when evaluating more than one predicate. Hence, the work inside the loop body and the number of predicates are important characteristics that have to be investigated to find the optimal scan implementation.

In this paper, we contribute a reasonable 3-dimensional evaluation of the impact of *the work inside the loop body*, the impact of *different selectivities for two predicates*, as well as the impact of *the number of predicates* on the performance of a scan pipeline including aggregations. As a result, we will derive guidelines when to use which scan implementation variant to gain the optimal performance.

The remainder of the paper is structured as follows. In Section 2, we present possible code optimizations from the literature that can be applied to a scan operator. Then, we evaluate the resulting scan pipelines using three different scenarios in Section 3. Finally, we present related work in Section 4 and conclude in Section 5.

2 Variants for Predicate Evaluation

There are several ways to optimize a selection operator. In this section, we will review three implementation strategies of single-predicate selections. Furthermore, when considering more than one predicate, we also have to care about how to combine the predicates, which adds the possibility for another variant to be considered for a query compiler. As a side note, all these optimizations are also applicable for code that is not produced by a query compiler. However, implementing one variant per predicate type, number of predicates, and aggregate creates a huge implementation effort and will probably avoided in practice.

2.1 Single-Predicate Variants

There are three implementation strategies for a single-predicate scan. They include to use an `if`-clause, which results in a *branching variant*, to use *predication*, which results in a branch-free variant, and to use *SIMD* to accelerate the predicate evaluation. Furthermore, a SIMD can be implemented using an `if`-clause or predication, which adds another possible variant. Notably, using a multi-threaded implementation is another optimization for scans. However, multi-threading adds a constant speedup to the before-mentioned variants without changing the overall performance differences between the three implementation strategies [BBS14].

Branching Scan. The code that is generated for a branching scan is pretty simple. The pipeline itself contains a loop iterating over the input to which the branching scan is adding an `if`-clause with the predicate evaluation. Inside the `if`-clause, subsequent operators will put their code. In our example in Listing 1, we show the branching scan for a `less than` predicate and an aggregation on a specific column that is executed depending on the result of the predicate evaluation. However, this implementation will only benefit from very high selectivities (i.e., only a small amount of tuples match), because else the CPU incurs many branch mispredictions causing considerable performance penalties [BBS14; RBZ13].

```
1  for(int i = 0; i < input_size; ++i){
2    if(col[i] < pred)
3        agg+=agg_col[i];
4  }
```

List. 1: Branching scan for `less than` predicate adapted from [BBS14].

```
1  for(int i = 0; i < input_size; ++i){
2    agg+=agg_col[i]*(col[i] < pred);
3  }
```

List. 2: Predicated scan for `less than` predicate adapted from [BBS14].

```
1  for(int i = 0; i < simd_size; ++i){
2    mask= SIMD_COMP(simd_col[i],pred);
3    if(mask){
4      for (int j=0;j < SIMD_LENGTH;++j){
5        if((mask >> j) & 1)
6          agg+=agg_col[i];
7      }
8    }
9  }
```

List. 3: SIMD scan for `less than` predicate adapted from [BBS14].

Predicated Scan. The predicated scan omits an if-clause and will always write a result. For this, we can use in C++ the return value of the comparison to manipulate the written result [RBZ13; Ro04]. The comparison will produce 1, if the condition is true, and 0 if the condition is false. This trick can help for creating an RID list [BBS14], but also for manipulating an aggregate computation. For this, you just multiply the aggregate with the result of the predicate, which will mask the aggregate value if the condition is false. We show the resulting code for a predicated aggregating scan in Listing 2.

SIMD Scan. *Single Instruction Multiple Data* is a technology that enables to execute one instruction on several data items in parallel. Since most of the processor's control logic depends on the number of in-flight instructions and registers, but not on the size of the registers, a theoretical speedup of factor n for a vector size of n elements is possible. However, this is hardly achieved in practice. For instance, the SIMD scan cannot benefit from the full potential, as it is impossible to do branching on a single data item in a SIMD

fashion. To perform a selection on a single data item, a mask has to be extracted and depending on the bits that are set, the branching is executed in a scalar fashion [ZR02]. We visualize the code of the branching SIMD scan in Listing 3. The computation of the mask is done in Line 2, that includes a SIMD macro for the predicate evaluation as well as a macro for extracting the mask. Afterwards, the mask is checked for at least one match and from Line 4 to Line 7 the single bits of the mask are examined. In case of a match, the aggregation is executed.

Predicated SIMD Scan. Another option is to fully avoid branches in the SIMD scan. In this variant, we use a bitwise `AND` between the result of the SIMD predicate evaluation and the aggregate column to mask mismatching values of the aggregate column. To get the final result, all entries of the SIMD register have to be summed up at last. Due to space reasons, we refer the interested programmer to the code in our GIT repository[4].

2.2 Multi-Predicate Variants

To evaluate multiple predicates, we just have to extend the above variants to evaluate not one, but several predicates. However, how to combine the predicates in an `if`-clause opens another tuning opportunity, because we can use a conditional `AND` or a bitwise `AND`. Notably, these two variations only apply to the branching scan, because (1) the predicated omits every branch and, thus, will only use the bitwise variant and (2) the SIMD variants only support a bitwise `AND` as macro.

Conditional `AND`. The conditional `AND` (e.g., $pred_1$ && $pred_2$) between predicates will start to evaluate the first predicate at first and only if it evaluates to true, the second predicate will be evaluated. This is also often called a short-circuit evaluation, because the computation of further predicates can be skipped, when the first predicate is already evaluated to false. So, it yields a speedup, if the first predicate is very selective [Ro04]. However, each conditional `AND` will produce a conditional branch in the execution and may lead to heavy branch misprediction penalties.

Bitwise `AND`. The bitwise `AND` (e.g., $pred_1$ & $pred_2$) inside an `if`-clause will evaluate all predicates, form the final result, and then execute the branch according to the result of the predicate evaluation. In this way, branch misprediction penalties are reduced (except the last one for the `if`-clause), but it misses the possibility to skip irrelevant predicates as the conditional `AND` can do.

3 Performance Evaluation of Scan Variants

In this section, we are presenting the performance differences of our scan variants for the three different impact factors that can be derived from the motivational example in Figure 1. The impact factors are the *work* done inside the loop, the *selectivity differences* between several predicates, and also the *number of predicates* to be evaluated. The scans

[4] http://git.iti.cs.ovgu.de/dbronesk/BTW-Pipeline-Variants

were implemented in C++ and compiled with the GCC 5.1 All experiments are executed on an Intel Xeon E5-2630 v3 using a single thread. As a side note, parallelized variants would show the same behavior as the single-threaded variants only with a meaningful speedup. The scanned data is taken from the TPC-H Benchmark and uses the `LineItem`[5] table of scale factor 10. Our predicates are `less then` predicates evaluated on independent columns of the table and predicate values are chosen from the data such that we meet the required selectivities.

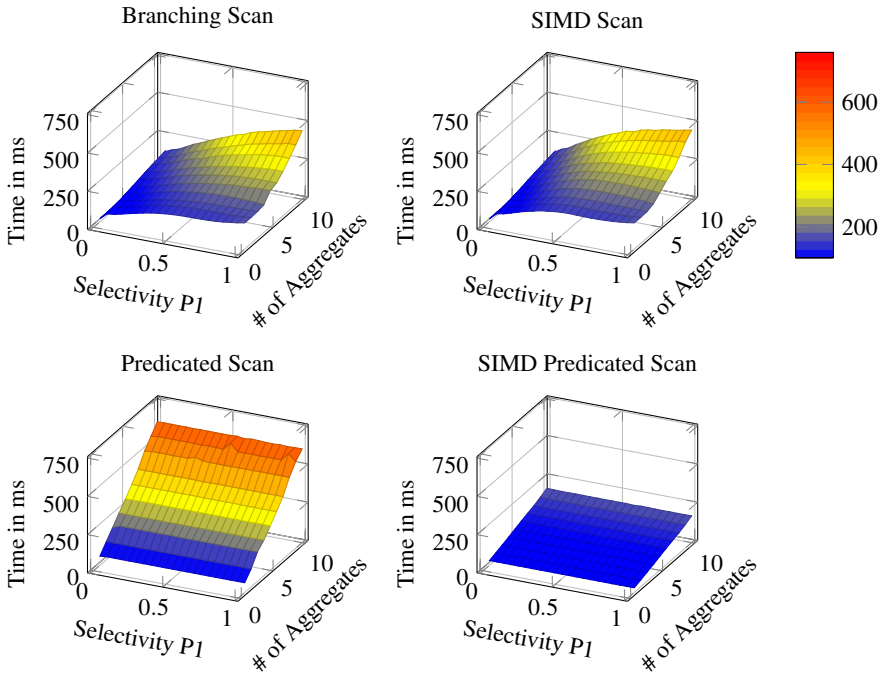

Fig. 2: Variant response time under different numbers of aggregates.

3.1 Single-Predicate Variants under Differing Workload Inside the Loop

For the evaluation of the impact of differing workload inside the loop, we emulate a pipeline with a single predicate and varied the number of aggregates inside the loop. For each aggregate, we have a separate field in the output object and we sum up a unique column per aggregate. Notably, other aggregate functions will only add a different overhead to the execution, but the overall behavior of the variants will stay the same. For instance, the aggregate function `count()` will issue minimal memory access cost (or none), because the aggregate will probably be held in the CPU register and is only incremented. In contrast, holistic aggregate functions (e.g., median) will at first gather all matching values, which will create two memory accesses (for reading the input and for writing the output

[5] The `LineItem` table of scale factor 10 contains 60 million tuples, such that a scan will touch 240 MB of data, which exceeds cache sizes by far. Hence, a bigger scale factor would lead to the same performance behavior.

array). Nevertheless, especially the SIMD predicated version is not applicable for grouped aggregations, as it would use a scatter operation. Hence, this is open for future work.

Since we are only varying the workload inside the loop in this experiment, we only use the four single-predicate variants: branching scan, predicated scan, SIMD scan, and predicated SIMD scan. We visualize the response time of our variants for different numbers of aggregates in Figure 2.

Branching Variants. We can derive from the plots for the branching variants that the selectivity factor has a big impact on the performance only if the number of aggregates is small. The more aggregates we take, the more linear is the progression of the curve. This means, that the overhead for a branch misprediction is hidden by other computations. Overall, the branching scan outperforms the SIMD scan except for selectivity factors around 0, where SIMD has a performance benefit of 13 %.

Predicated Variants. The predicated variants show no impact of the selectivity factor, but a high impact of the number of aggregates, which is consistent with our expectation. However, we can see, that the predicated SIMD scan consistently outperforms the predicated scan by a factor between 2 and 3.5.

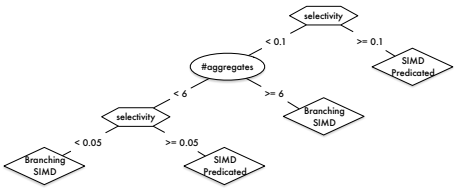

Fig. 3: Decision tree for several aggregates.

Overall Comparison. Comparing the branching variants with the predicated variants, we can see that the selectivity factor range, in which the predicated variants are better than the branching variants, becomes smaller the more aggregates have to be computed. In particular, the branching SIMD scan outperforms the SIMD predicated scan for selectivity factors smaller than 0.05 for up to five aggregates, while the threshold is at 0.1 for more than five aggregates (cf. Figure 3).

3.2 Multi-Predicate Variants for Differing Predicate Selectivities

In this experiment, we want to examine the impact of different selectivities of two predicates on the multi-predicate variants. The evaluated variants are: the branching scan using conditional `AND`, the branching scan using bitwise `AND`, the predicated scan, the SIMD scan, and the predicated SIMD scan. For a good comparison, we use a single aggregation inside the loop, because it is a case, where the predicated variants should still show considerable performance compared to the branching variants. This experiment is especially interesting for the two new variants of the branching variant, because the conditional `AND` is sensitive to the single selectivities, while the other branching variants are only sensitive to the accumulated selectivity. Notably, a query optimizer would order the predicates on their selectivity such that the first predicate would be the most selective. Hence, only the half of the parameter space that we use in this experiment applies for real-world scenarios.

Branching Variants. If we compare the branching variants, we can see that our expectations were partially met. The conditional `AND` scan shows a high performance dependence on the first predicate, while the SIMD branching scan shows a symmetric behavior w.r.t. the order of the predicates (i.e., it does not matter whether P1 is more selective than P2, or the other way around). However, the bitwise `AND` scan still shows a lazy evaluation, which is caused by a bad compiler optimization. Still, if the first predicate has a selectivity factor around 0, the conditional `AND` scan outperforms the other two branching scans by up to factor 1.4. However, if both predicates are very selective reaching a small combined selectivity factor higher than 0.05, the SIMD scan outperforms the conditional `AND` scan.

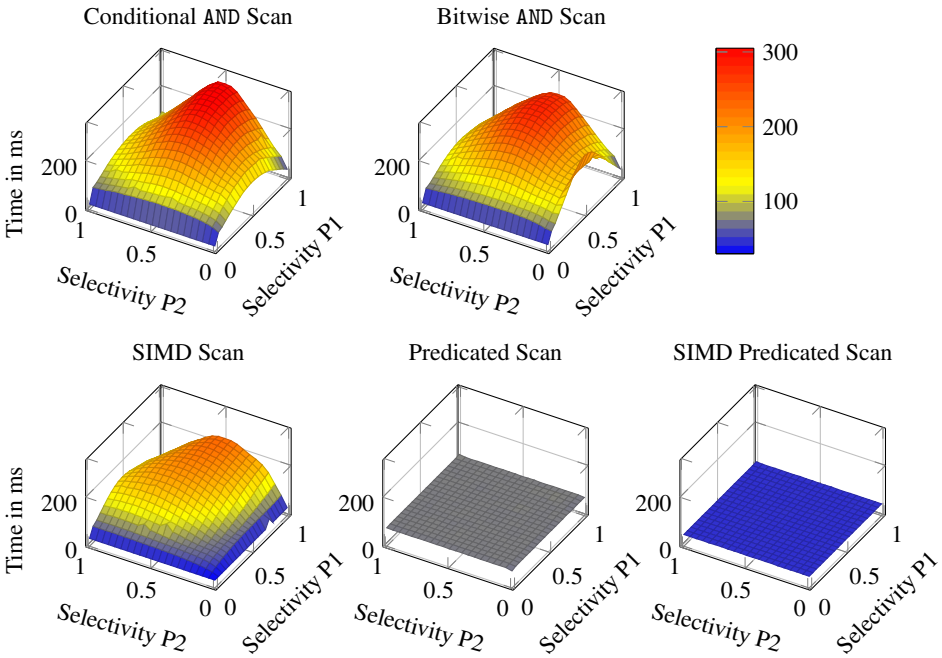

Fig. 4: Variant response time under different selectivities of two predicates.

Predicated Variants. For the predicated variants, we can see that the selectivity factor has no impact on the variant performance, as the resulting performance graph is a plane parallel to the x-y plane. Comparing both predicated variants, the SIMD predicated scan outperforms the predicated scan by a factor of 1.6 in all cases.

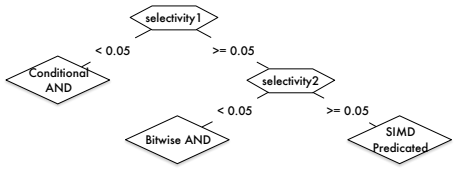

Fig. 5: Decision tree for different selectivities.

Overall Comparison. When comparing branching and predicated variants, we can see that if the predicate with the highest selectivity has a selectivity factor higher than 0.05, the predicated SIMD scan performs better than the branching scans (cf. Figure 5). Else, one of the branching scans performs best depending on the single selectivities.

In summary, the decisions of a query compiler for real-world scenarios depend on the selectivity of the predicate with the highest selectivity due to the predicate ordering during query optimization. If the first predicate has a selectivity close to 0, the conditional AND scan will be chosen by the query compiler. In any other case, the SIMD predicated scan will be chosen.

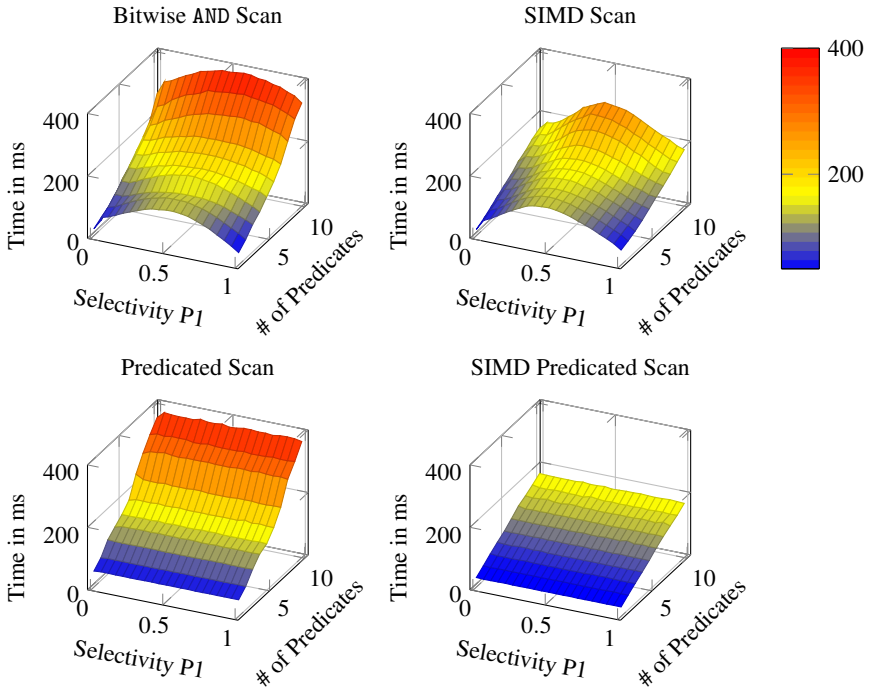

Fig. 6: Variant response time for different numbers of predicates.

3.3 Multi-Predicate Variants for Differing Number of Predicates

In this experiment, we want to examine the impact of an increasing number of predicates on the performance of our multi-predicate variants, especially looking at the difference between SIMD and non-SIMD variants. For the experiment, we exclude the conditional AND scan, because we cannot visualize the different selectivities of more than two predicates. In fact, we keep the accumulated selectivity factor ranges constant while stepwise adding more predicates in order to increase the work of predicate evaluation. Similar to the experiment before, we use a single aggregate as work inside the loop.

Branching Variants. Similar to the increase of work inside the loop, an increasing number of predicates will also smoothen the curve of branching scans along the selectivity factor, although this is more evident for the bitwise AND scan than for the SIMD scan. This is due to the small overhead of the branch misprediction when compared to the overall response time. Furthermore, when comparing the SIMD scan with the bitwise AND scan, we can see that

the bitwise AND scan does not scale as good as the SIMD scan with an increasing number of predicates and it can only partially outperform the SIMD scan for a small number of predicates. In fact, for up to four predicates, the bitwise AND scan outperforms the SIMD scan for selectivity factors in the range [0.1,0.65]. However for more than four predicates, the SIMD scan constantly outperforms the bitwise AND scan, because it can reuse the results inside the SIMD registers.

Predicated Variants. The predicated variants show no impact of the selectivity factor on their performance. However, the number of predicates adds a constant factor to the response time, which is higher for the predicated non-SIMD variant than for the predicated SIMD variant. Similar to the branching variants, for a *single* predicate, the non-SIMD variant performs better than the SIMD variant, while it is the other way around for more than one predicate.

Overall Comparison. When comparing the branching and the predicated variants, we can see that for selectivity factors close to 0, the branching variants outperform their predicated counterparts. For selectivity factors close to 0, performance factors from 1.23 to 1.64 are possible. However, for a selectivity factor between [0.05,0.95], the predicated variants reach a benefit of up to factor of 2.1 in comparison to the best performing branching variant.

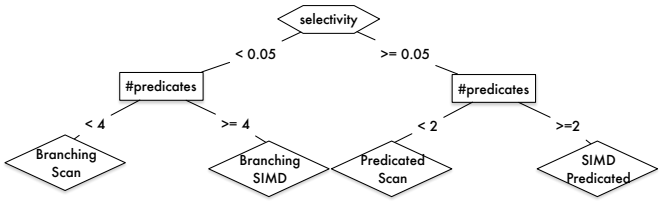

Fig. 7: Decision tree for several predicates.

As shown in Figure 7, a query compiler should use a branching scan for accumulated selectivity factors less than 0.05. In this case, the branching scan is chosen for less than 4 predicates while, otherwise, the SIMD scan is used. For other selectivity factors, the query compiler should use a predicated variant – it should use a predicated SIMD variant for more than one predicate and for one predicate, it should rely on a non-SIMD variant.

4 Related Work

Using SIMD to accelerate database operations, in particular selection conditions, has gained much attention in research. The first idea of a SIMD scan, which is also our basis for the SIMD scan, was proposed by Zhou and Ross [ZR02]. Later on, Polychroniou and Ross extend the work to AVX by using bloom filters [PR14], Sitaridi and Ross to exploit GPUs [SR13] and Polychroniou et al. to exploit Xeon Phis [PRR15]. Furthermore, Willhalm et al. use compression and SIMD acceleration to speed up the scan for single [Wi09] and complex [Wi13] predicates. All these improvements propose reasonable variants that could be used in future work to extend the findings of this paper and could be merged into a cost model for query compilers.

5 Conclusion

In our paper, we compare the performance of different scan pipeline implementations w.r.t. increasing work inside the loop, differing selectivities of two predicates, and increasing numbers of predicates. Our evaluation has shown, that all parameters have a high impact on the usage of the variants and a query compiler should take care of these parameters to select the right variant in order to produce the best performing pipeline code. Nevertheless, exact thresholds still depend on the used machine and have to be re-evaluated for every new CPU architecture.

References

[Ba13] Balkesen, C.; Teubner, J.; Alonso, G.; Özsu, M. T.: Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In: ICDE. IEEE Computer Society, pp. 362–373, 2013.

[BBS14] Broneske, D.; Breß, S.; Saake, G.: Database scan variants on modern CPUs: A performance study. In: VLDB Workshop IMDM. Vol. 8921. LNCS, Springer, pp. 97–111, 2014.

[Br14] Broneske, D.; Breß, S.; Heimel, M.; Saake, G.: Toward hardware-sensitive database operations. In: EDBT. Pp. 229–234, 2014.

[Ne11] Neumann, T.: Efficiently compiling efficient query plans for modern hardware. PVLDB 4/9, pp. 539–550, 2011.

[PR14] Polychroniou, O.; Ross, K.: Vectorized bloom filters for advanced SIMD processors. In: SIGMOD Workshop DaMoN. ACM, 2014.

[PRR15] Polychroniou, O.; Raghavan, A.; Ross, K. A.: Rethinking SIMD vectorization for in-memory databases. In: SIGMOD. ACM, pp. 1493–1508, 2015.

[RBZ13] Răducanu, B.; Boncz, P.; Zukowski, M.: Micro adaptivity in Vectorwise. In: SIGMOD. Pp. 1231–1242, 2013.

[Ro04] Ross, K. A.: Selection conditions in main-memory. In: TODS. Vol. 29. 1, pp. 132–161, 2004.

[SR13] Sitaridi, E.; Ross, K.: Optimizing select conditions on GPUs. In: SIGMOD Workshop DaMoN. ACM, 4:1–4:8, 2013.

[Tr14] Transaction Processing Performance Council: TPC BENCHMARK H (Decision Support), tech. rep. 2.17.1, 2014.

[Wi09] Willhalm, T.; Boshmaf, Y.; Plattner, H.; Popovici, N.; Zeier, A.; Schaffner, J.: SIMD-Scan: Ultra fast in-memory table scan using on-chip vector processing units. PVLDB 2/1, pp. 385–394, 2009.

[Wi13] Willhalm, T.; Oukid, I.; Müller, I.; Faerber, F.: Vectorizing database column scans with complex predicates. In: VLDB Workshop ADMS. Pp. 1–12, 2013.

[ZR02] Zhou, J.; Ross, K. A.: Implementing database operations using SIMD instructions. In: SIGMOD. Pp. 145–156, 2002.