Vectorization vs. Compilation in Query Execution

Juliusz Sompolski[1]
VectorWise B.V.
julek@vectorwise.com

Marcin Zukowski
VectorWise B.V.
marcin@vectorwise.com

Peter Boncz[2]
Vrije Universiteit Amsterdam
p.a.boncz@vu.nl

ABSTRACT

Compiling database queries into executable (sub-) programs provides substantial benefits comparing to traditional interpreted execution. Many of these benefits, such as reduced interpretation overhead, better instruction code locality, and providing opportunities to use SIMD instructions, have previously been provided by redesigning query processors to use a *vectorized execution model*. In this paper, we try to shed light on the question of how state-of-the-art compilation strategies relate to vectorized execution for analytical database workloads on modern CPUs. For this purpose, we carefully investigate the behavior of vectorized and compiled strategies inside the Ingres VectorWise database system in three use cases: Project, Select and Hash Join. One of the findings is that compilation should always be combined with block-wise query execution. Another contribution is identifying three cases where "loop-compilation" strategies are inferior to vectorized execution. As such, a careful merging of these two strategies is proposed for optimal performance: either by incorporating vectorized execution principles into compiled query plans or using query compilation to create building blocks for vectorized processing.

1. INTRODUCTION

Database systems provide many useful abstractions such as data independence, ACID properties, and the possibility to pose declarative complex ad-hoc queries over large amounts of data. This flexibility implies that a database server has no advance knowledge of the queries until runtime, which has traditionally led most systems to implement their query evaluators using an interpretation engine. Such an engine evaluates plans consisting of algebraic *operators*, such as Scan, Join, Project, Aggregation and Select. The operators internally include *expressions*, which can be boolean conditions used in Joins and Select, calculations used to introduce new columns in Project, and functions like MIN, MAX and SUM used in Aggregation. Most query interpreters follow the so-called iterator-model (as described in Volcano [5]), in which each operator implements an API that consists of open(), next() and close() methods. Each next() call produces one new tuple, and query evaluation follows a "pull" model in which next() is called recursively to traverse the operator tree from the root downwards, with the result tuples being pulled upwards.

It has been observed that the tuple-at-a-time model leads to interpretation overhead: the situation that much more time is spent in evaluating the query plan than in actually calculating the query result. Additionally, this tuple-at-a-time interpretation model particularly affects high performance features introduced in modern CPUs [13]. For instance, the fact that units of actual work are hidden in the stream of interpreting code and function calls, prevents compilers and modern CPUs from getting the benefits of deep CPU pipelining and SIMD instructions, because for these the work instructions should be adjacent in the instruction stream and independent of each other.

Related Work: Vectorized execution. MonetDB [2] reduced interpretation overhead by using *bulk processing*, where each operator would fully process its input, and only then invoking the next execution stage. This idea has been further improved in the X100 project [1], later evolving into VectorWise, with *vectorized execution*. It is a form of block-oriented query processing [8], where the next() method rather than a single tuple produces a block (typically 100-10000) of tuples. In the vectorized model, data is represented as small single-dimensional arrays (vectors), easily accessible for CPUs. The effect is (i) that the percentage of instructions spent in interpretation logic is reduced by a factor equal to the vector-size, and (ii) that the functions that perform work now typically process an array of values in a tight loop. Such tight loops can be optimized well by compilers, e.g. unrolled when beneficial, and enable compilers to generate SIMD instructions automatically. Modern CPUs also do well on such loops, as function calls are eliminated, branches get more predictable, and out-of-order execution in CPUs often takes multiple loop iterations into execution concurrently, exploiting the deeply pipelined resources of modern CPUs. It was shown that vectorized execution can improve data-intensive (OLAP) queries by a factor 50.

Related Work: Loop-compilation. An alternative strategy for eliminating the ill effects of interpretation is using Just-In-Time (JIT) query compilation. On receiving a query

[1]This work is part of a MSc thesis being written at Vrije Universiteit Amsterdam.
[2]The author also remains affiliated with CWI Amsterdam.

Proceedings of the Seventh International Workshop on Data Management on New Hardware (DaMoN 2011), June 13, 2011, Athens, Greece.

for the first time, the query processor compiles (part of) the query into a routine that gets subsequently executed. In Java engines, this can be done through the generation of new Java classes that are loaded using reflection (and JIT compiled by the virtual machine) [10]. In C or C++, source code text is generated, compiled, dynamically loaded, and executed. System R originally skipped compilation by generating assembly directly, but the non-portability of that approach led to its abandonment [4]. Depending on the compilation strategy, the generated code may either solve the whole query ("holistic" compilation [7]) or only certain performance-critical pieces. Other systems that are known to use compilation are ParAccel [9] and the recently announced Hyper system [6]. We will generalise the current state-of-the-art using the term "loop-compilation" strategies, as these typically try to compile the core of the query into a single loop that iterates over tuples. This can be contrasted with vectorized execution, which decomposes operators in multiple basic steps, and executes a separate loop for each basic step ("multi-loop").

Compilation removes interpretation overhead and can lead to very concise and CPU-friendly code. In this paper, we put compilation in its most favourable light by assuming that compilation-time is negligible. This is often true in OLAP queries which tend do be rather long-running, and technologies such as JIT in Java and the LLVM framework for C/C++ [12] nowadays provide low (milliseconds) latencies for compiling and linking.

Roadmap: vectorization vs. compilation. Vectorized expressions process one or more input arrays and store the result in an output array. Even though systems like VectorWise go through lengths to ensure that these arrays are CPU cache-resident, this materialization constitutes extra load/store work. Compilation can avoid this work by keeping results in CPU registers as they flow from one expression to the other. Also, compilation as a general technique is orthogonal to any execution strategy, and can only improve performance. We used the VectorWise DBMS[3] to investigate three interesting use cases that highlight the issues around the relationship between compilation and vectorization.

As our first case, Section 2 shows how in Project expression calculations loop-compilation tends to provide the best results, but that this hinges on using block-oriented processing. Thus, compiling expressions in a tuple-at-a-time engine may improve some performance, but falls short of the gains that are possible. In Section 3, our second case is Select, where we show that branch mispredictions hurt loop-compilation when evaluating conjunctive predicates. In contrast, the vectorized approach avoids this problem as it can transform control-dependencies into data-dependencies for evaluating booleans (along [11]). The third case in Section 4 concerns probing large hash-tables, using a HashJoin as an example. Here, loop-compilation gets CPU cache miss stalled while processing linked lists (i.e., hash bucket chains). We show that a mixed approach that uses vectorization for chain-following is fastest, and robust to the various parameters of the key lookup problem. These findings lead to a set of conclusions which we present in Section 5.

[3]See `www.ingres.com/vectorwise`. Data storage and query evaluation in VectorWise is based on the X100 project [1].

Algorithm 1 Implementation of an example query using vectorized and compiled modes. Map-primitives are statically compiled functions for combinations of operations (OP), types (T) and input formats (col/val). Dynamically compiled primitives, such as c000(), follow the same pattern as pre-generated vectorized primitives, but may take arbitrarily complex expressions as OP.

```
// General vectorized primitive pattern
map_OP_T_col_T_col(idx n,T* res,T* col1,T* col2){
  for(int i=0; i<n; i++)
    res[i]=OP(col1[i],col2[i]);
}

// The micro-benchmark uses data stored in:
const idx LEN=1024;
chr tmp1[LEN], tmp2[LEN], one = 100;
sht tmp3[LEN];
int tmp4[LEN];  // final result

// Vectorized code:
map_add_chr_val_chr_col(LEN,tmp1,&one,l_discount);
map_sub_chr_val_chr_col(LEN,tmp2,&one,l_tax);
map_mul_chr_col_chr_col(LEN,tmp3,tmp1,tmp2);
map_mul_int_col_sht_col(LEN,tmp4,l_extprice,tmp3);

// Compiled equivalent of this expression:
c000(idx n,int *res,int *col1,chr* col2,chr* col3){
  for(idx i=0; i<n; i++)
    res[i]=col1[i]*((100-col2[i])*(100+col3[i]));
}
```

2. CASE STUDY: PROJECT

Inspired by the expressions in Q1 of TPC-H we focus on the following simple Scan-Project query as micro-benchmark:

```
SELECT l_extprice*(1-l_discount)*(1+l_tax) FROM lineitem
```

The scanned columns are all decimals with precision two. VectorWise represents these internally as integers, using the value multiplied by 100 in this case. After scanning and decompression it chooses the smallest integer type that, given the actual value domain, can represent the numbers. The same happens for calculation expressions, where the destination type is chosen to be the minimal-width integer type, such that overflow is prevented. In the TPC-H case, the price column is a 4-byte integer and the other two are single-byte columns. The addition and subtraction produce again single bytes, their multiplication a 2-byte integer. The final multiplication multiplies a 4-byte with a 2-byte integer, creating a 4-byte result.

Vectorized Execution. VectorWise executes functions inside a Project as so-called map-primitives. Algorithm 1 shows the example code for a binary primitive. In this, `chr`, `sht`, `int` and `lng` represent internal types for 1-, 2-, 4- and 8-byte integers and `idx` represents an internal type for representing sizes, indexes or offsets within columns of data (implemented as integer of required width). A `_val` suffix in the primitive name indicates a constant (non-columnar) parameter. VectorWise pre-generates primitives for all needed combinations of operations, types and parameter modes. All functions to support SQL fit in ca. 9000 lines of macro-code, which expands into roughly 3000 different primitive functions producing 200K LOC and a 5MB binary.

It is reasonable to expect that a compiler that claims support for SIMD should be able to vectorize the trivial loop in the `map_` functions. On x86 systems, gcc (we used 4.5.1) usually does so and the Intel compiler icc never fails to. With

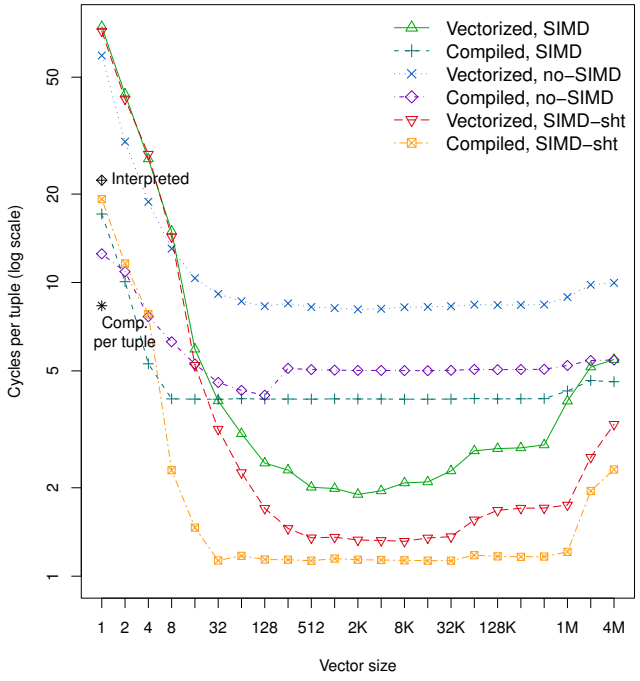

Figure 1: Project micro-benchmark: with and without {compilation, vectorization, SIMD}. The "SIMD-sht" lines work around the alignment sub-optimality in icc SIMD code generation.

a *single* SSE instruction, modern x86 systems can add and subtract 16 single-byte values, multiply 8 single-byte integers into a 2-byte result, or multiply four 4-byte integers. Thus, 16 tuples in this expression could be processed with 8 SIMD instructions: one 8-bit addition, one 8-bit subtraction, two 8-bit multiplications with 16-bit results, and four 32-bit multiplications. All of these instructions store one result and the first two operations load one input (with the other parameter being a constant) while the other two load two inputs. With these 22 (2*2+6*3) load/stores, we roughly need 30 instructions – in reality some additional instructions for casts, padding and looping are required, such that the total for processing 16 tuples is around 60. In comparison, without SIMD we would need 4 instructions (2 loads, 1 calculation, 1 store) per calculation such that a single tuple requires 16 instructions, > 4 times more than with SIMD.

The vectorized "SIMD" and "no-SIMD" lines in Figure 1, show an experiment in which expression results are calculated, using different vector-sizes. We used a 2.67GHz Nehalem core, on a 64-bits Linux machine with 12GB of RAM. The no-SIMD vectorized code, produced by explicitly disabling SIMD generation in the compiler (icc 11.0[4], here), is indeed 4 times slower than SIMD. The general trend of decreasing interpretation overhead with increasing vector-size until around one thousand, and performance deteriorating due to cache misses if vectors start exceeding the L1 and L2 caches, has been described already in detail in [13, 1].

Compiled Execution. The lower part of Algorithm 1 shows the compiled code that a modified version of VectorWise can now generate on-the-fly: it combines vectorization with compilation. Such a combination in itself is not new ("compound primitives" [1]), and the end result is similar to what a holistic query compiler like HIQUE [7] generates for this Scan-Project plan, though it would also add Scan code. However, if we assume a HIQUE with a simple main-memory storage system and take `l_tax`, etc. to be pointers into a column-wise stored table, then `c000()` would be the exact product of a "loop-compilation" strategy.

The main benefit of the compiled approach is the absence of load/stores. The vectorized approach needs 22 load/stores, but only the bottom three loads and top-level store are needed by the compiled strategy. Comparing vectorized with compiled, we are surprised to see that the vectorized version is significantly faster (4 vs. 2 cycles/tuple). Close investigation of the generated code revealed that icc chooses in its SIMD generation to align all calculations on the widest unit (here: 4-byte integers). Hence, the opportunities for 1-byte and 2-byte SIMD operations are lost. Arguably, this is a compiler bug or sub-optimality.

In order to show what compilation could achieve, we retried the same, now assuming that `l_extprice` would fit into a 2-byte integer; which are the "SIMD-sht" lines in in Figure 1. Here, we see compilation beating vectorized execution, as one would normally expect in Project tasks. A final observation is that compiled map-primitives are less sensitive to cache size (only to L2, not L1), such that a hybrid vectorized/compiled engine can use large vector-sizes.

Tuple-at-a-time compilation. The black star and diamond in Figure 1, correspond to situations where primitive functions work tuple-at-a-time. The non-compiled strategy is called "interpreted", here. An engine like MySQL, whose whole iterator interface is tuple-at-a-time, can only use such functions as it has just one tuple to operate on at any moment in time. Tuple-at-a-time primitives are conceptually very close to the functions in Algorithm 1 with vector-size `n=1`, but lack the for-loop. We implemented them separately for fairness, because these for-loops introduce loop-overhead. This experiment shows that if one would contemplate introducing compilation in an engine like MySQL without breaking its tuple-at-a-time operator API, the gain in expression calculation performance could be a factor 3 (23 vs 7 cycle/tuple). The absolute performance is clearly below what block-wise query processing offers (7 vs 1.2cycle/tuple), mainly due to missed SIMD opportunities, but also because the virtual method call for every tuple inhibits speculative execution across tuples in the CPU. Worse, in tuple-at-a-time query evaluation function primitives in OLAP queries only make up a small percentage (<5%) of overall time [1], because most effort goes into the tuple-at-a-time operator APIs. The overall gain of using compilation without changing the tuple-at-a-time nature of a query engine can therefore at most be a few percent, making such an endeavour questionable.

3. CASE STUDY: SELECT

We now turn our attention to a micro-benchmark that tests conjunctive selections:

```
WHERE col1 < v1 AND col2 < v2 AND col3 < v3
```

Selection primitives shown in Algorithm 2 create vectors of indexes for which the condition evaluates to true, called

[4]Compiler options are -O3 for gcc, supplemented for icc with -xSSE4.2 -mp1 -unroll

Algorithm 2 Implementations of < selection primitives. All algorithms return the number of selected items (return j). For mid selectivities, branching instructions lead to branch mispredictions. In a vectorized implementation such branching can be avoided. VectorWise dynamically selects the best method depending on the observed selectivity, but in the micro-benchmark we show the results for both methods.

```
// Two vectorized implementations
// (1.) medium selectivity: non-branching code
idx sel_lt_T_col_T_val(idx n,T*res,T*col1,T*val2,
                                              idx*sel){
  if(sel== NULL) {
    for(idx i=0, idx j=0; i<n; i++) {
      res[j] = i; j += (col1[i] < val2[0]);
    }
  } else {
    for(idx i=0, idx j=0; i<n; i++) {
      res[j] = sel[i]; j += (col1[sel[i]] < *val2);
    }
  }
  return j;
}
// (2.) else: branching selection
idx sel_lt_T_col_T_val(idx n,T*res,T*col1,T*val2,
                                              idx*sel){
  if(sel==NULL) {
    for(idx i=0, idx j=0; i<n; i++)
      if(col1[i] < *val2) res[j++] = i;
  } else {
    for(idx i=0, idx j=0; i<n; i++)
      if(col1[sel[i]] < *val2) res[j++] = sel[i];
  }
  return j;
}

// Vectorized conjunction implementation:
const idx LEN=1024;
idx sel1[LEN],sel2[LEN],res[LEN],ret1,ret2,ret3;
ret1 = sel_lt_T_col_t_val(LEN,sel1,col1,&v1,NULL);
ret2 = sel_lt_T_col_t_val(ret1,sel2,col1,&v1,sel1);
ret3 = sel_lt_T_col_t_val(ret2,res,col1,&v1,sel2);
```

selection vectors. Selection primitives can also take a selection vector as parameter, to evaluate the condition only on elements of the vectors from the positions pointed to by the selection vector [5]. A vectorized conjunction is implemented by chaining selection primitives with the output selection vector of the previous one being the input selection vector of the next one, working on a tightening subset of the original vectors, evaluating this conjunction lazily only on those elements for which the previous conditions passed.

Each condition may be evaluated with one of two implementations of selection primitive. The naive "branching" implementation of selection evaluates conditions lazily and branches out if any of the predicates fails. If the selectivity of conditions is neither very low or high, CPU branch predictors are unable to correctly guess the branch outcome. This prevents the CPU from filling its pipelines with useful future code and hinders performance. In [11] it was shown that a branch (control-dependency) in the selection code can be transformed into a data dependency for better performance.

The `sel_lt` functions in Algorithm 2 contain both approaches. The VectorWise implementation of selections uses a mechanism that chooses either the branch or non-branch

[5]In fact, other primitives are also able to work with selection vectors, but it was removed from code snippets where not necessary for the discussed experiments.

Algorithm 3 Four compiled implementations of a conjunctive selection. Branching cannot be avoided in loop-compilation, which combines selection with other operations, without executing these operations eagerly. The four implementations balance between branching and eager computation.

```
// (1.) all predicates branching ("lazy")
idx c0001(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(i=0; i<n; i++)
    if (col1[i]<*v1 && col2[i]<*v2 && col3[i]<*v3)
      res[j++] = i;
  return j; // return number of selected items.
}

// (2.) branching 1,2, non-br. 3
idx c0002(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(j=0; i<n; i++)
    if (col1[i]<*v1 && col2[i] < *v2) {
      res[j] = i; j += col3[i] < *v3;
    }
  return j;
}

// (3.) branching 1, non-br. 2,3
idx c0003(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i,j=0;
  for(i=0; i<n; i++)
    if (col1[i]<v1) {
      res[j] = i; j += col2[i]<*v2 & col3[i]<*v3
    }
  return j;
}

// (4.) non-branching 1,2,3, ("compute-all")
idx c0004(idx n,T* res,T* col1,T* col2,T* col3,
                              T* v1, T* v2, T* v3) {
  idx i, j=0;
  for(i=0; i<n; i++) {
    res[j] = i;
    j += (col1[i]<*v1 & col2[i]<*v2 & col3[i]<*v3)
  }
  return j;
}
```

strategy depending on the observed selectivity [6]. As such, its performance achieves the minimum of the vectorized branching and non-branching lines in Figure 2.

In this experiment, each of the columns `col1`, `col2`, `col3` is an integer column, and the values `v1`, `v2` and `v3` are constants, adjusted to control the selectivity of each condition. Here, we keep the selectivity of each branch equal, hence to the cube root of the overall selectivity, which we vary from 0 to 1. We performed the experiment on 1K input tuples.

Figure 2 shows that compilation of conjunctive Select is inferior to the pure vectorized approach. The lazy compiled program does slightly outperform vectorized branching, but for the medium selectivities branching is by far not the best option. The gist of the problem is that the trick of (i) converting all control dependencies in data dependencies while still (ii) avoiding unnecessary evaluation, cannot be achieved in a single loop. If one avoids all branches (the "compute-all" approach in Algorithm 3), all conditions always get evaluated, wasting resources if a prior condition already failed.

[6]It even re-orders dynamically the conjunctive predicates such that the most selective is evaluated first.

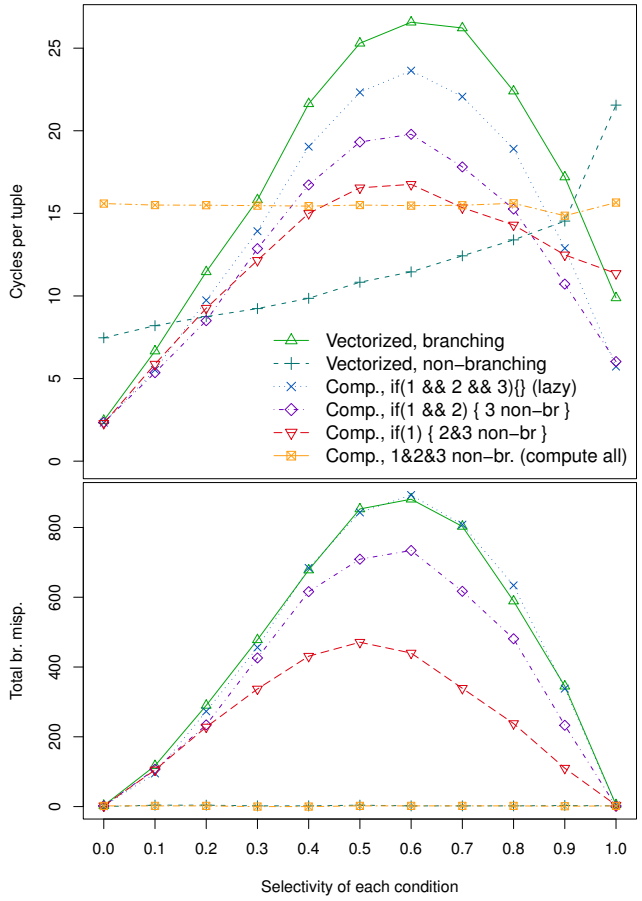

Figure 2: Conjunction of selection conditions: total cycles and branch mispredictions vs. selectivity

One can try mixed approaches, branching on the first predicates and using data dependency on the remaining ones. They perform better in some selectivity ranges, but maintain the basic problems – their worst behavior is when the selectivity after branching predicates is around 50%.

4. CASE STUDY: HASH JOIN

Our last micro-benchmark concerns Hash Joins:

```
SELECT build.col1, build.col2, build.col3
WHERE probe.key1 = build.key1 AND probe.key2 = build.key2
FROM probe, build
```

We focus on an equi-join condition involving keys consisting of two (integer) columns, because such composite keys are more challenging for vectorized executors. This discussion assumes simple bucket-chaining, such as used in VectorWise, presented in Figure 3. This means that keys are hashed on buckets in an array B with size N which is a power of two. Each bucket contains the offset of a tuple in a value space V. This space can either be organized using DSM or NSM layout; VectorWise supports both [14]. It contains the values of the build relation, as well as a next-offset, which implements the bucket chain. A bucket may have a chain of length > 1 either due to hash collisions, or because there are multiple tuples in the build relation with the same key.

Algorithm 4 Vectorized implementation of hash probing.

```
map_hash_T_col(idx n, idx* res, T* col1){
  for(idx i=0; i<n; i++)
    res[i] = HASH(col1[i]);
}
map_rehash_idx_col_T_col(idx n, idx* res,
        idx* col1, T* col2) {
  for (idx i=0; i<n; i++)
    res[i] = col1[i] ^ HASH(col2[i]);
}
map_fetch_idx_col_T_col(idx n, T* res,
        idx* col1, T* base, idx* sel){
  if(sel) {
    for (idx i=0;i<n; i++)
      res[sel[i]] = base[col1[sel[i]]];
  } else {/* sel == NULL, omitted */}
}
map_check_T_col_idx_col_col_T_col(idx n, chr* res,
        T* keys, T* base, idx* pos, idx* sel) {
  if(sel) {
    for(idx i=0; i<n; i++)
      res[sel[i]] =
            (keys[sel[i]]!=base[pos[sel[i]]]);
  } else {/* sel == NULL, omitted */}
}
map_recheck_chr_col_T_col_idx_col_T_col(idx n,
        chr* res, chr* col1,
        T* keys, T* base, idx* pos, idx* sel) {
  if(sel) {
    for(idx i=0; i<n; i++)
      res[sel[i]] = col1[sel[i]] ||
            (keys[sel[i]]!=base[pos[sel[i]]]);
  } else {/* sel == NULL, omitted */}
}
ht_lookup_initial(idx n, idx *pos, idx* match,
        idx* H, idx* B) {
  for(idx i=0,k=0; i<n; i++) {
    // saving found chain head position in HT
    pos[i] = B[H[i]];
    // saving to a sel. vector if non-zero
    if(pos[i]) { match[k++] = i; }
  }
}
ht_lookup_next(idx n,idx* pos,idx* match,
        idx* next) {
  for(idx i=0,k=0; i<n; i++) {
    // advancing to next in chain
    pos[match[i]] = next[pos[match[i]]];
    // saving to a sel. vec. if non-empty
    if(pos[match[i]]) { match[k++] = match[i]; }
  }
}
```

procedure HTPROBE($V, B[0..N-1], \vec{K_{1..k}}$**(in)**, $\vec{R_{1..v}}$**(out)**)
  // Iterative hash-number computation
  $\vec{H} \leftarrow map\_hash(\vec{K_1})$
  for each remaining key vectors K_i **do**
    $\vec{H} \leftarrow map\_rehash(\vec{H}, \vec{K_i})$
  $\vec{H} \leftarrow map\_bitwiseand(\vec{H}, N-1)$
  // Initial lookup of candidate matches
  $\vec{Pos}, \vec{Match} \leftarrow ht\_lookup\_initial(H, B)$
  while $\vec{Match}$ not empty **do**
    // Candidate value verification
    $\vec{Check} \leftarrow map\_check(\vec{K_1}, V_{key_1}, \vec{Pos}, \vec{Match})$
    for each remaining key vector $\vec{K_i}$ **do**
      $\vec{Check} \leftarrow map\_recheck(\vec{Check}, \vec{K_i}, V_{key_i}, \vec{Pos}, \vec{Match})$
    $\vec{Match} \leftarrow sel\_nonzero(\vec{Check}, \vec{Match})$
    // Chain following
    $\vec{Pos}, \vec{Match} \leftarrow ht\_lookup\_next(\vec{Pos}, \vec{Match}, V_{next})$
  $\vec{Hits} \leftarrow sel\_nonzero(\vec{Pos})$
  // Fetching the non-key attributes
  for each result vector $\vec{R_i}$ **do**
    $\vec{R_i} \leftarrow map\_fetch(\vec{Pos}, V_{value_i}, \vec{Hits})$

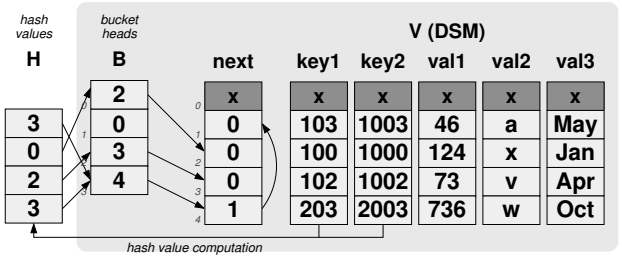

Figure 3: Bucket-chain hash table as used in VectorWise. The value space V presented in the figure is in DSM format, with separate array for each attribute. It can also be implemented in NSM, with data stored tuple-wise.

Vectorized Hash Probing. For space reasons we only discuss the probe phase in Algorithm 4, we show code for the DSM data representation and we focus on the scenario when there is at most one hit for each probe tuple (as is common with relations joined with a foreign-key referential constraint). Probing starts by vectorized computation of a hash number from a key in a column-by-column fashion using map-primitives. A `map_hash_T_col` primitive first hashes each key of type `T` onto a `lng` long integer. If the key is composite, we iteratively refine the hash value using a `map_rehash_lng_col_T_col` primitive, passing in the previously computed hash values and the next key column. A bitwise-and map-primitive is used to compute a bucket number from the hash values: $H\&(N\text{-}1)$.

To read the positions of heads of chains for the calculated buckets we use a special primitive `ht_lookup_initial`. It behaves like a selection primitive, creating a selection vector $\vec{Match}$ of positions in the bucket number vector H for which a match was found. Also, it fills the $\vec{Pos}$ vector with positions of the candidate matching tuples in the hash table. If the value (offset) in the bucket is 0, there is no key in the hash table – these tuples store 0 in $\vec{Pos}$ and are not part of $\vec{Match}$.

Having identified the indices of possible matching tuples, the next task is to "check" if the key values actually match. This is implemented using a specialized map primitive that combines fetching a value by offset with testing for non-equality: `map_check`. Similar to hashing, composite keys are supported using a `map_recheck` primitive which gets the boolean output of the previous check as an extra first parameter. The resulting booleans mark positions for which the check failed. The positions can then be selected using a select `sel_nonzero` primitive, overwriting the selection vector $\vec{Match}$ with positions for which probing should advance to the "next" position in the chain. Advancing is done by a special primitive `ht_lookup_next`, which for each probe tuple in $\vec{Match}$ fills $\vec{Pos}$ with the next position in the bucket-chain of V. It also guards for ends of chain by reducing $\vec{Match}$ to its subset for which the resulting position in $\vec{Pos}$ was non-zero.

The loop finishes when the $\vec{Match}$ selection vector becomes empty, either because of reaching end of chain (element in $\vec{Pos}$ equals 0, a miss) or because checking succeeded (element in $\vec{Pos}$ pointing to a position in V, a hit).

Hits can be found by selecting the elements of $\vec{Pos}$ which ultimately produced a match with a `sel_nonzero` primitive.

Algorithm 5 Fully loop-compiled hash probing: for each NSM tuple, read hash bucket from B, loop through a chain in V, fetching results when the check produces a match

```
for(idx i=0, j=0; i<n; i++) {
  idx pos, hash = HASH(key1[i])^HASH(key2[i]);
  if (pos = B[hash&(N-1)]) do {
    if(key1[i]==V[pos].key1 &&
       key2[i]==V[pos].key2) {
        res1[i] = V[pos].col1;
        res2[i] = V[pos].col2;
        res3[i] = V[pos].col3;
        break; // found match
    }
  } while(pos = V.next[pos]); // next
}
```

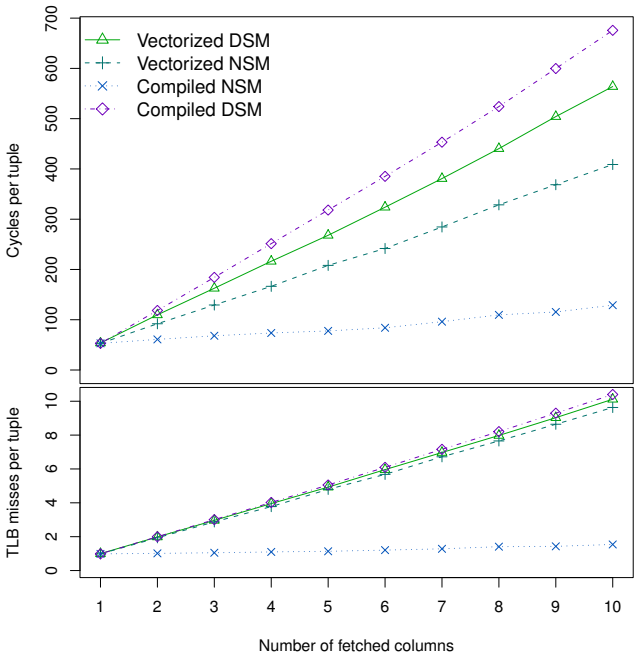

Figure 4: Fetching columns of data from a hash table: cycles per tuple and total TLB misses

$\vec{Pos}$ with selection vector $\vec{Hits}$ becomes a pivot vector for fetching. This pivot is subsequently used to fetch (non-key) result values from the build value area into the hash join result vectors; using one fetch primitive for each result column.

Partial Compilation. There are three opportunities to apply compilation to vectorized hashing. The first is to compile the full sequence of hash/rehash/bitwise-and and bucket fetch into a single primitive. The second combines the check and iterative re-check (in case of composite keys) and the select > 0 into a single select-primitive. Since the test for a key in a well-tuned hash table has a selectivity around 75%, we can restrict ourselves to a non-branching implementation. These two opportunities re-iterate the compilation benefits of Project and Select primitives, as discussed in the previous sections, so we omit the code.

The third opportunity is in the fetch code. Here, we can generate a composite fetch primitive that, given a vector of positions, fetches multiple columns. The main benefit of this

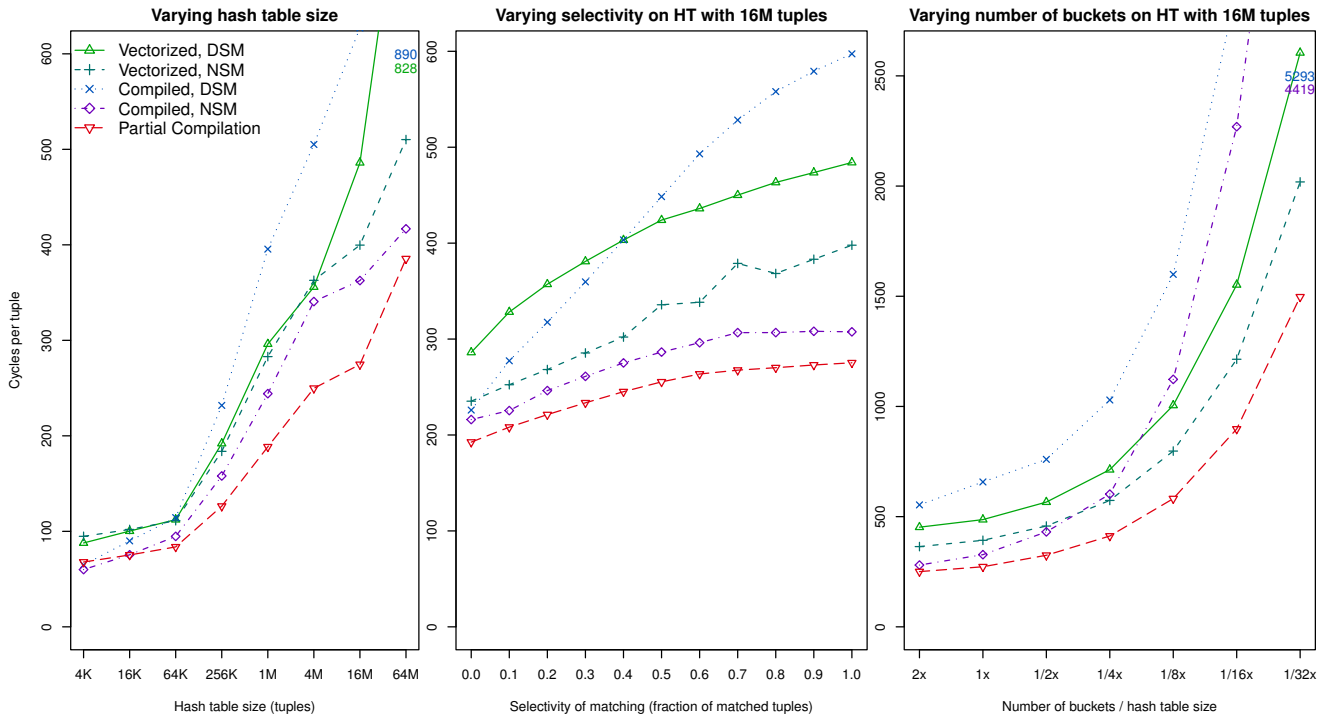

Figure 5: Hash table probing. Left: different sizes of the hash table value space V. Middle: different match rates of the probe. Right: different sizes of the bucket array B w.r.t. value space V (different chain lengths).

is obtained in case of NSM organisation of the value space V. Vectorization fetches values column-at-a-time, hence passes over the NSM value space as many times as there are result columns (here: 3), accessing random positions from the value space on each pass. On efficient vector-sizes, the amount of random accesses is surely larger than TLB cache size and may even exceed the amount of cache lines, such that TLB and cache trashing occurs, with pages and cache lines from the previous pass being already evicted from the caches before the next. The compiled fetch fetches all columns from the same position at once, achieving more data locality. Figure 4 shows that in normal vectorized hashing performance of NSM and DSM is similar, but compilation makes NSM clearly superior.

Full Compilation. It is possible to create a loop-compiled version of the full join, like e.g. proposed in HIQUE [7]. Algorithm 5 shows the hash probe section of such an algorithm, which we also tested. It loops over probe keys, and for each probe key fetches the corresponding bucket, then iterates over the bucket-chain, checking the key for equality, and if equal, fetches the needed result columns. We try to explain in the following why this fully compiled algorithm is inferior to the vectorized alternative with partial compilation.

Parallel Memory Access. Because memory latency is not improving much (∼100ns), and cache line granularities must remain limited (64bytes), memory bandwidth on modern hardware is no longer simply the division between these two. A single Nehalem core can achieve a factor 10 more than this 0.64GB/s, thanks to automatic CPU prefetching on sequential access. Performance thus crucially depends on having multiple outstanding memory requests at all times. For random access, this is hard to achieve, but the deeply pipelined out-of-order nature of modern CPUs does help. That is, if a load stalls, the CPU might be able to speculate ahead into upstream instructions and reach more loads. The Intel Nehalem architecture can have four outstanding loads, improving bandwidth by a factor four[7]. Success is not guaranteed, since the instruction speculation window of a CPU is limited, depends on branch prediction, and only independent upstream instructions can be taken into execution.

The Hard-To-Understand Part. The crux here is that the vectorized fetch primitive trivially achieves whatever maximum outstanding loads the CPU supports, as it is a tight loop of independent loads. The same holds for the partially compiled variants. The fully compiled hash probe, however, can run aground while following the bucket chain. Its performance is only good if the CPU can speculate ahead across multiple probe tuples (execute concurrently instructions from multiple for-loop iterations on `i`). That depends on the branch predictor predicting the `while(pos..)` to be false, which will happen in join key distributions where there are no collisions. If, however, there are collisions or if the build relation has multiple identical keys, the CPU will stall with a single outstanding memory request (`V[pos]`), because the branch predictor will make it stay in the while-loop, and it will be unable to proceed as the value of `pos = V.next[pos]` is unknown because it depends on the current cache/TLB miss. A similar effect has been described in the context of using explicit prefetching instructions in hash-joins [3]. This

[7]Speculation-friendly code is thus more effective than manual prefetching, which tends to give only minor improvement, and is hard to tune/maintain for multiple platforms.

effect causes fully compiled hashing to be four times slower than vectorized hashing in the worst case.

Experiments. Figure 5 shows experiments for hash probing using the vectorized, fully and partially compiled approaches, using both DSM and NSM as the hash table (V) representation. We vary hash table size, selectivity (fraction of probe keys that match something), and bucket chain length; which have default values resp. 16M, 1.0 and 1. The left part shows that when the hash table size grows, performance deteriorates; it is well understood that cache and TLB misses are to blame, and DSM achieves less locality than NSM. The middle graph shows that with increasing hit rate, the cost goes up, which mainly depends on increasing fetch work for tuple generation. The compiled NSM fetch alternatives perform best, as explained. The right graph shows what happens with increasing chain length. As discussed above, the fully compiled (NSM) variant suffers most, as it gets no parallel memory access. The overall best solution is partially compiled NSM, thanks to its efficient compiled multi-column fetching (and to a lesser extent efficient checking/hashing, in case of composite keys) and its parallel memory access, during lookup, fetching and chain-following.

5. CONCLUSIONS

For database architects seeking a way to increase the computational performance of a database engine, there might seem to be a choice between vectorizing the expression engine versus introducing expression compilation. Vectorization is a form of block-oriented processing, and if a system already has an operator API that is tuple-at-a-time, there will be many changes needed beyond expression calculation, notably in all query operators as well as in the storage layer. If high computational performance is the goal, such deep changes cannot be avoided, as we have shown that if one would keep adhering to a tuple-a-time operator API, expression compilation alone only provides marginal improvement.

Our main message is that one does not need to choose between compilation and vectorization, as we show that best results are obtained if the two are combined. As to what this combining entails, we have shown that "loop-compilation" techniques as have been proposed recently can be inferior to plain vectorization, due to better (i) SIMD alignment, (ii) ability to avoid branch mispredictions and (iii) parallel memory accesses. Thus, in such cases, compilation should better be split in multiple loops, materializing intermediate vectorized results. Also, we have signaled cases where an interpreted (but vectorized) evaluation strategy provides optimization opportunities which are very hard with compilation, like dynamic selection of a predicate evaluation method or predicate evaluation order.

Thus, a simple compilation strategy is not enough; state-of-the art algorithmic methods may use certain complex transformations of the problem at hand, sometimes require run-time adaptivity, and always benefit from careful tuning. To reach the same level of sophistication, compilation-based query engines would require significant added complexity, possibly even higher than that of interpreted engines. Also, it shows that vectorized execution, which is an evolution of the iterator model, thanks to enhancing it with compilation further evolves into an even more efficient and more flexible solution without making dramatic changes to the DBMS architecture. It obtains very good performance while maintaining clear modularization, simplified testing and easy performance- and quality-tracking, which are key properties of a software product.

6. REFERENCES

[1] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *Proc. CIDR*, Asilomar, CA, USA, 2005.

[2] P. A. Boncz. *Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications*. Ph.d. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands, May 2002.

[3] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *Proc. ICDE*, Boston, MA, USA, 2004.

[4] D. Chamberlin et al. A history and evaluation of System R. *Commun. ACM*, 24(10):632–646, 1981.

[5] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE TKDE*, 6(1):120–135, 1994.

[6] A. Kemper and T. Neumann. HyPer: Hybrid OLTP and OLAP High Performance Database System. Technical report, Technical Univ. Munich, TUM-I1010, May 2010.

[7] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[8] S. Padmanabhan, T. Malkemus, R. Agarwal, and A. Jhingran. Block Oriented Processing of Relational Database Operations in Modern Computer Architectures. In *Proc. ICDE*, Heidelberg, Germany, 2001.

[9] ParAccel Inc. Whitepaper. *The ParAcel Analytical Database: A Technical Overview*, Feb 2010. `http://www.paraccel.com`.

[10] J. Rao, H. Pirahesh, C. Mohan, and G. M. Lohman. Compiled Query Execution Engine using JVM. In *Proc. ICDE*, Atlanta, GA, USA, 2006.

[11] K. A. Ross. Conjunctive selection conditions in main memory. In *Proc. PODS*, Washington, DC, USA, 2002.

[12] The LLVM Compiler Infrastructure . `http://llvm.org`.

[13] M. Zukowski. *Balancing Vectorized Query Execution with Bandwidth-Optimized Storage*. Ph.D. Thesis, Universiteit van Amsterdam, Sep 2009.

[14] M. Zukowski, N. Nes, and P. Boncz. DSM vs. NSM: CPU Performance Tradeoffs in Block-Oriented Query Processing. 2008.