Assessing and Optimizing Microarchitectural Performance of Event Processing Systems

Marcelo R.N. Mendes, Pedro Bizarro, Paulo Marques

CISUC, University of Coimbra,
Dep. Eng. Informática – Pólo II, 3030-290, Coimbra, Portugal
{mnunes, bizarro, pmarques}@dei.uc.pt

Abstract. Event Processing (EP) systems are being progressively used in business critical applications in domains such as algorithmic trading, supply chain management, production monitoring, or fraud detection. To deal with high throughput and low response time requirements, these EP systems mainly use the CPU-RAM sub-system for data processing. However, as we show here, collected statistics on CPU usage or on CPU-RAM communication reveal that available systems are poorly optimized and grossly waste resources. In this paper we quantify some of these inefficiencies and propose cache-aware algorithms and changes on internal data structures to overcome them. We test the before and after system both at the micro-architecture and application level and show that: i) the changes improve micro-architecture metrics such as clocks-per-instruction, cache misses or TLB misses; ii) and that some of these improvements result in very high application level improvements such as a 44% improvement on stream-to-table joins with 6-fold reduction on memory consumption, and order-of-magnitude increase on throughput for moving aggregation operations.

Keywords: Benchmarking, Complex Event Processing, Performance, Tuning.

1 Introduction

Previous work by Ailamaki [2], Ravishankar [11], and Abadi [1] showed that micro-architecture inspired improvements such as cache-aware algorithms and changes of internal data representations can lead to high improvements on the performance of database or data warehouse systems. Encouraged by this work, we took a similar position and set-out to discover the micro-architecture performance of Event Processing (EP) systems.

Using Esper [6], a widely used open-source EP system, we measured the systems' performance executing common operations such as moving aggregations and stream-to-table joins. We monitored the system at the micro-architecture level using the *Intel VTune*® profiler [8] and also collected application-level metrics such as memory consumption and peak sustained throughput.

To isolate from secondary effects, we then replicated the main algorithms and data structures on our own event processing prototype and progressively improved them with microarchitecture-aware optimizations. These optimizations were then validated first by running the tuned prototype in a multi-query scenario, and then porting the modifications back into Esper.

Summary of Contributions

In this paper we make the following contributions:

- We analyzed how current event processing systems perform at both application and hardware levels. By collecting and correlating metrics such as throughput, CPI, and cache misses during execution of continuous queries, we show that micro-architectural aspects significantly influence the final performance of common event processing tasks, being in some cases the sole cause for performance degradation when the input is scaled up (Section 3.3).
- We demonstrated how alternate data structures can drastically improve performance and resource consumption in EP systems (Section 3.4).
- We implemented, tested and evaluated an adapted version of the Grace Hash algorithm [9] for joining event streams with memory-resident tables. Results revealed that by reducing the impact of micro-architectural aspects on query execution performance was improve in up to 44 percent (Section 3.5).
- We implemented, tested and evaluated a microarchitecture-aware algorithm for computing moving aggregations over sliding windows, which provided performance gains ranging from 28 to 35 percent (Section 3.5).

2 Background

From the microarchitectural point of view, the amount of time a given computational task *T* takes to complete depends primarily on two factors: the task size (i.e., Instruction Count or IC) and the average duration of instructions (frequently expressed as Cycles per Instruction or *CPI*). Algebraically, in cycles [7]:

$$CPU\ execution\ time = IC \times CPI$$

Better performance can be achieved by reducing either factor or both. Traditionally, software developers have focused on reducing IC by improving time complexity of algorithms, but an increased interest in making a more efficient use of hardware resources has been observed over the last years [2, 3, 12, 13].

To understand how these optimizations targeted at the hardware level work, it is necessary to know the internals of CPU operation. Every single instruction is executed inside the processor as series of sequential steps across its several functional units. During this sequence of steps, generally referred as *pipeline*, CPU instructions are fetched from memory, decoded, executed and finally have their results stored back into registers or memory. To increase throughput, instructions in different stages/functional units are processed in parallel (*Instruction-Level Parallelism*). In ideal conditions, the processor pipeline remains full most of the time, retiring one or more instructions per cycle (implying CPI ≤ 1). Many factors, however, can cause instructions to *stall*, thus increasing average CPI. The gap between processor and memory speeds is one of them: an instruction that access data resident in main memory may require tens to hundreds of CPU cycles to complete. Another typical cause of stalls is *data dependency*: when an instruction *j* depends on a value produced by an instruction *i* that was fetched closely before it, *j* cannot execute until *i* completes (in fact, if the processor issues instructions in program order, when instruction *j* stalls, no later instructions can proceed, thus aggravating the performance impact of the data dependency). Finally, *control dependencies*, which happen when the instruction flow cannot be determined until a given instruction *i* (e.g., a conditional branch) completes, can also adversely affect the degree of instruction-level parallelism achieved.

In order to attenuate the aforementioned stalls, hardware vendors have devised several techniques. For example, to minimize memory-related stalls, smaller and faster *cache* memories are placed in the data path between processor and main memory. The strategy is to benefit from the locality principle and serve most memory requests with data coming from lower-latency cache accesses. Additionally, data dependencies are minimized by allowing instructions to execute out-of-order inside the pipeline. Finally, control dependencies are partially addressed via speculative execution (i.e., the processor executes instructions that lie beyond a conditional branch as if it had been already resolved).

In practice, the characteristics of the applications determine whether the hardware techniques above will be successful or not at making the processor execute close to its full capacity. With that in mind, a number of novel analytical databases had been developed over the last years, in an attempt to better exploit the internal features of processors. Examples of microarchitectural optimizations employed by such databases include a column-oriented data organization and compression techniques which together provide a more efficient use of memory hierarchy [12, 13]. Also, recently proposed compression algorithms [14] minimize the negative impact of branch mispredictions by removing `if-then-else` constructs from their critical path.

We argue that similar optimizations can be applied in the context of event processing systems, resulting in potentially higher gains since in this case data is manipulated mostly in main memory.

3 Assessing and Improving CPU Performance of EP Systems

This section introduces the workload, benchmark setup and an evaluation of Esper and of our own prototype with and without CPU and cache-aware optimizations.

3.1 Workload

In order to assess the gains of the optimizations proposed in this work we used a simple, though representative workload, composed by two common operations performed by event processing systems: moving aggregations over event streams and correlation of event streams with historic data (*join*). These two queries and the dataset are described in detail next.

Dataset

Input data consisted in a generic event stream *S* and table *T* with schemas shown in Figure 1 and Figure 2.

```
CREATE STREAM S (
    ID integer,
    A1 double,
    A2 double,
    TS long)
```

Figure 1: Schema of Input Dataset (Stream "S")

In our tests the values assumed by `S` attributes are not relevant for query performance evaluation, so they were filled with a fixed, pre-generated, data value – *this ensures that measurements are minimally affected by data generation.* The exception is the attribute `ID`, which is used to join stream `S` with table `T`. In this case, `S`'s `ID` was filled with random values uniformly distributed in the range of `T`'s `ID`.

```
CREATE TABLE T (
    ID integer,
    T1 integer,
    T2 integer,
    T3 integer,
    T4 integer)
```

Figure 2: Schema of table "T" used in the join query

`T`'s `ID` attribute, used to perform the join with the event stream, assumes unique values ranging from 1 to `TABLE_SIZE_IN_ROWS`. The other four attributes in `T` do not influence query performance and are filled with random data.

Aggregation Query

The aggregation query used in this study computes a moving average over a *count-based sliding* window[1]. The query is shown below using syntax of CQL [4], a SQL-based query language for continuous stream processing:

```
SELECT avg(A1)
FROM S [ROWS N Slide 1]
```

Figure 3: Aggregation query, written in CQL

For this particular query, every event arrival at stream `S` causes the output of an updated result. Parameter `N` represents the window size, which varies across the tests, ranging from 1000 to 100 million events.

[1] "*Count-based*" means that the window size is defined in terms of the number of events over which the computation takes place. "*Sliding*" means that once the window is full events are expired one-by-one upon arrival of new events.

Join Query

To examine the behavior of an event processing system when performing a join, we used a query based on a real use-case of a telecom company that needed to join streaming call detail records (CDR) (here represented by stream S) with historic data (represented by table T). In our tests this query is expressed as follows:

```
SELECT S.ID, S.A1, T.T1
FROM   S, T
WHERE  S.ID = T.ID
```

Figure 4: Join query – stream S with table T

Since the goal here is to focus on the performance of processor and memory hierarchy, the table is maintained in main memory, thus eliminating eventual effects of the I/O subsystem on the results. (*As pointed out before, keeping the dataset in memory is commonplace in most EP applications, especially those which require high processing throughputs and/or low latencies.*) The selectivity of the query is always 100% (i.e., every event is matched against one and only one record in the table) and the table size is varied across tests, ranging from 1 thousand to 10 million rows.

3.2 Experiments: Setup and Methodology

All the tests were carried out on a server with two Intel Xeon E5420 Quad-Core processors (Core® microarchitecture, L2-Cache: 12MB, 2.50 GHz, 1333 MHz FSB), 16 GB of RAM, running Windows Server 2008 x64 and Sun Hotspot x64 JVM.

The performance measurements were done as follows:

- A single Java application was responsible for generating, submitting and consuming tuples during the performance runs. Events are submitted and processed through local method calls, so that measurements are not affected by network/communication effects.
- In the tests with join queries, load generation was preceded by an initial loading phase, during which the in-memory table was populated with a given number of records.
- Load generation started with an initial 1-minute warmup phase, with events (S tuples) being generated and consumed at the maximum rate supported.
- Warmup was followed by a 15-minute measurement phase, during which we collected both application-level metrics and hardware-level metrics. Application-level metrics, namely throughput and memory consumption, were gathered inside the Java application. Throughput was computed as total event count divided by elapsed time. Memory consumption was computed using `totalMemory()` and `freeMemory(),` standard Java SDK `Runtime` class methods. Hardware-level metrics were obtained using the *Intel VTune*® profiler [8], as described in detail next. As in the warmup phase, events were submitted at the maximum rate sustained by the specific implementation.
- Each test was repeated 3 times and the reported metrics were averaged.

VTune collects metrics by inspecting specific hardware counters provided by Intel processors. To avoid excessive monitoring overhead, only one or two events are collected at a time – for doing that, VTune breaks event collection in "runs". In our tests, each run has a duration of 2 minutes (1 minute for calibration and 1 minute for counters collection), and 6 runs were necessary to collect all the configured metrics.

We started the experimental evaluation by measuring the performance of the two queries in the open-source EP engine Esper (Section 3.3) and then implemented the same basis algorithms employed by it on a separate, custom-code, Java prototype. These baseline algorithms were then progressively modified to include the proposed optimizations while we performed new measurements[2] (Sections 3.4 and 3.5). Finally, we validated the optimizations by measuring their performance in more realistic scenarios: first with the prototype processing several queries simultaneously (Section 3.6) and then with them being inserted into Esper code (Section 3.7).

The basis algorithm for computing the aggregation query works as follows. The *sliding window* is implemented as a circular buffer, internally represented as a fixed-length array; the average aggregation itself is computed by updating count and sum state variables upon event arrival/expiration. The join algorithm is also straightforward: it keeps the table into a hash index structure with the join attribute as key and then performs a lookup in that hash table every time a new event arrives.

3.3 Preliminary Results

Figure 5 illustrates the results for the join query running at the event processing engine Esper, using two different tuple representations: *Map* and *POJO* (in the former events are represented as instances of the standard `HashMap` Java class, while in the latter events are represented as fixed-schema Plain Java Objects)[3]:

Two important observations can be made from the graph above: First, for both tuple representations, the throughput dropped about 40 percent from a join with a 1000-rows table to a join with a 10M-rows table. This drop occurred even though the employed algorithm –

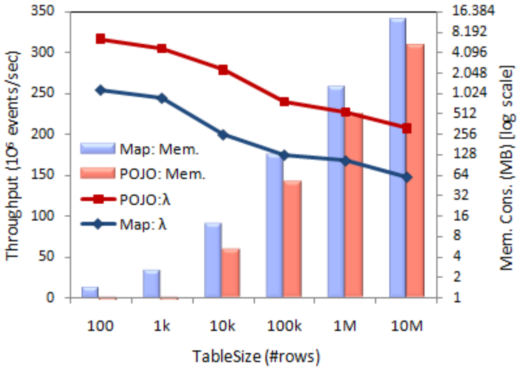


Figure 5: Performance of Join Query on Esper

[2] We could have implemented the changes only on Esper, but our goal was to assess the performance gains of the optimizations on query processing in spite of any product specificities. Besides, using a smaller custom-code application makes the effects of code changes more controllable and understandable.

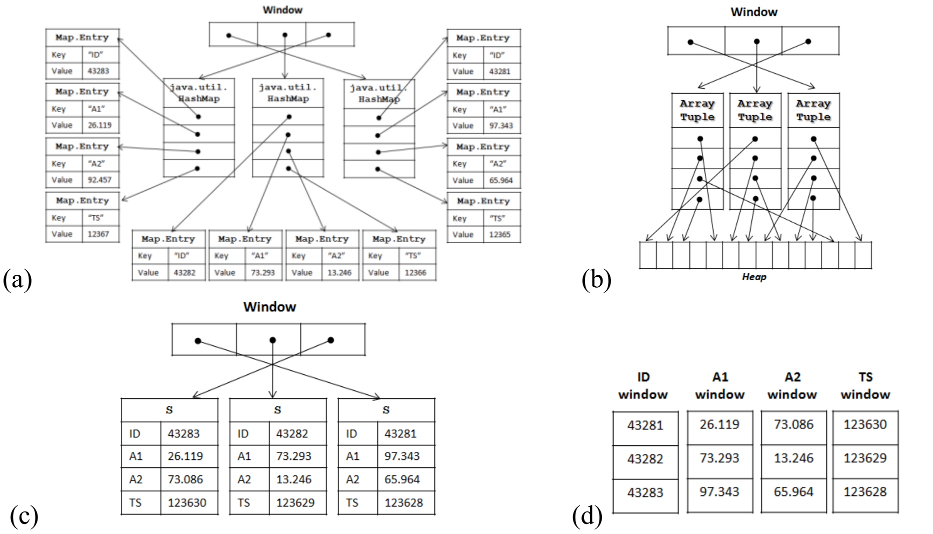
[3] In this work, we focus on Esper due to its open-source nature, which allowed us to port some of the optimizations here proposed into a real engine. We point out, however, that the results here presented are representative of a range of event processing engines, given that the behavior exhibited by Esper is similar to what was observed in previous tests with other products (see [10] for more details).

hash join – has a O(1) runtime complexity. Second, measured memory consumption during query execution was 4 to 60 times the space required for keeping the table data. Examining carefully the hardware metrics and Esper source code, we concluded that the drop on query performance was due mainly to microarchitectural aspects (and to a lesser extent, to increased garbage collection activity) and that the excessive memory consumption was caused by non-optimized internal representation of tuples as further explored in the next section.

3.4 Optimizing Internal Data Structures

In order to address the problem of excessive memory consumption, we focused first in optimizing the structures used to keep data items in main memory (i.e., the window for the aggregation query, and the table for the join query). Specifically, we were interested in finding out if the original representations used by Esper to represent the stream and table tuples could be improved and if a column-oriented storage model would result in enhanced performance in the context of event processing.

It is worthy to notice that column-store formats are especially useful for read-oriented, scan-oriented, non-ad-hoc queries. Thus, while on one hand EP systems, with their scan-oriented, non-ad-hoc queries may benefit from column-store representations, on the other hand, EP systems read/write workloads might be hurt by column-store representations.

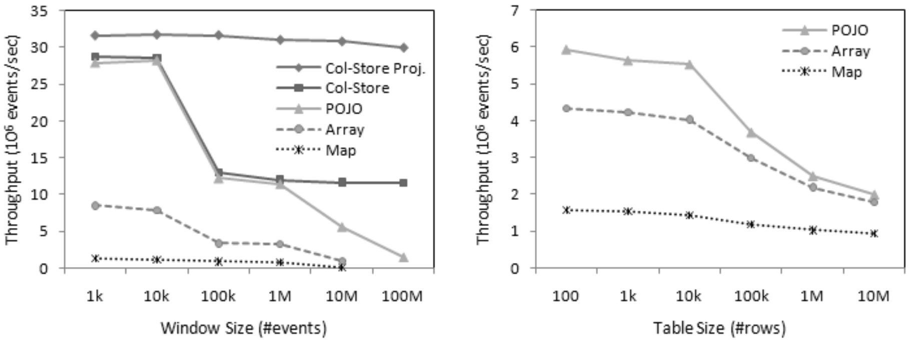


S		S		S	
ID	43283	ID	43282	ID	43281
A1	26.119	A1	73.293	A1	97.343
A2	73.086	A2	13.246	A2	65.964
TS	123630	TS	123629	TS	123628

ID window	A1 window	A2 window	TS window
43281	26.119	73.086	123630
43282	73.293	13.246	123629
43283	97.343	65.964	123628

Figure 6: The Different Data Structures used to represent tuples:
(a) Key-Value Map; (b) Array of Objects; (c) POJO; (d) Column-Store

To assess the impact of data structures on query performance, we started representing events/tuples as instances of the `HashMap` class (Figure 6(a)), and then employed progressively more lightweight representations: first as arrays of Objects (b), and then as fixed-schema Plain Java Objects (POJO) (c). Finally we tested the column-oriented storage model (d), in two different modalities: first keeping all original attributes of events/tuples (here named "*Col-Store*") and then keeping (projecting) only the attribute referenced in the query ("*Col-Store Proj.*"). In the *Col-Store* format, N aligned arrays of primitive types are kept in memory (where "N" is the number of attributes), while in the *Col-Store Proj* format only one array containing the values for the referenced attribute is maintained. Figure 7, Table 1 and Table 2 summarize the most relevant results for both the aggregation and join queries:



(a) (b)

Figure 7: Impact of Internal Representation on Performance: (a) Aggregation; (b) Join[4].

Table 1: Data Structures and Memory Consumption (in MB): Aggregation Query

Tuple Format	Window Size					
	1k	10k	100k	1M	10M	100M
Map	0.8	5.5	51.4	511.5	5,112.9	-
Array	0.5	2.1	17.9	176.5	1,755.0	-
POJO	0.3	0.8	6.3	61.2	682.1	6,103.0
Col-Store	0.2	0.5	2.8	26.9	267.2	2,670.5
Col-Store Proj.	0.2	0.3	0.9	7.8	76.0	763.1

Table 2: Data Structures and Memory Consumption (in MB): Join Query

Tuple Format	Table Size					
	100	1k	10k	100k	1M	10M
Map	0.2	1.1	9.3	92.1	920.0	9,131.2
Array	0.2	0.4	2.8	27.3	268.4	2,654.8
POJO	0.2	0.3	1.4	13.6	130.6	1,275.3
Col-Store	0.2	0.3	1.4	13.2	126.8	1,237.0
Col-Store Proj.	0.2	0.3	1.3	11.7	111.5	1,083.8

[4] For the sake of clarity, the lines "*Col-Store*" and "*Col-Store Proj*" were omitted in the Join graph since the results with these two implementations were very similar to the POJO case.

The numbers reveal that considerable gains in final performance and resource consumption could be obtained by using more optimized data structures. For the join query, the most efficient representation achieved more than 3 times more throughput than the original one, while consuming less than 15% memory space of what was originally required. The gains for the aggregation query were even more impressive: the best performing implementation (column-store keeping only required data) achieved on average about 35 times more throughput than the one using maps as event representation and reduced memory consumption to around 1,5% of the memory space occupied by this initial implementation.

Discussion: Aggregation Query

A couple of facts in the aggregation tests results are worth mentioning. First, using optimized data structures allowed the aggregation query to operate over windows of larger sizes, which otherwise would not be possible if employing the original non-optimized tuple representations (e.g., it was not possible to run the aggregation query over a window of 100M events when they were represented as Maps or arrays of Objects because these implementations required more memory space than it was physically available).

Second, the POJO format, although more efficient than Map and array representations, suffered severe drops in performance in two distinct points of the graph: from 10k to 100k and from 1M on. The collected metrics reveal that the first drop was caused by microarchitectural aspects (more specifically, an increase in L2 cache misses), while the second was due to an increased garbage collection activity.

Further results indicate that the column-oriented storage model addressed partially or even totally the aforementioned issues. For instance, in contrast with the *POJO* line, the *Col-Store* line in Figure 7-a remained steady after 1M. The reason for such difference is that the column-oriented representation, by using primitive types instead of Objects, is less susceptible to garbage collection than the POJO. In the best case (i.e., when keeping in memory only the attribute referenced by the query), the column-oriented model was also able to eliminate the microarchitectural issues. For that particular implementation, the memory access pattern is essentially sequential – consuming events from the stream and inserting them into the window means sequentially traversing a primitive type array – which maximizes performance at the microarchitectural level and ensures a steady throughput over the most different window sizes. Indeed, this observation was corroborated experimentally, with the CPI metric remaining basically unaffected in all tests of that specific implementation (ranged from 0.850 to 0.878 for windows of 1000 up to 100M events).

Discussion: Join Query

Interestingly, for the join query the column-oriented storage model did not provide considerable performance gains with respect to the POJO representation. (In fact, both throughput and memory consumption were quite similar in these two configurations and for this reason we omitted the results for column-store in Figure 7-b.) This behavior seems to be related to the fact that the amount of memory consumed by the payload itself (i.e., tuples attributes) in that case is small compared to the overhead of inherent factors of the Java programming environment (e.g., object alignment, wrappers for primitive types, and especially heavyweight native HashMap class, the structure used for indexing the table), which hides eventual gains provided by a more concise column-oriented dataset. Overall, the final performance of column-store implementation for the join query oscillated around +3% and -5% in comparison with the tests with POJO tuples.

3.5 Improving Algorithms Efficiency at the CPU Level

Nearly all results presented in previous section revealed that the throughput of both aggregation and join queries dropped significantly as the input size was increased, even though the theoretical runtime complexity of the employed algorithms is O(1). In this Section we delve into the causes for this behavior and propose optimizations to improve algorithms scalability with respect to input size.

Aggregation Query

As mentioned before, the algorithm for computing *subtractable* [5] aggregations such as AVG, SUM or COUNT over sliding windows consists essentially in updating some fixed set of state variables upon event arrival while maintaining the events of the window in main memory. As such, the algorithm has a O(N) space complexity but a theoretical O(1) time complexity. In practice, however, several factors can make the running time of the algorithm grow when the input size is increased. One of them is garbage collection: generally, the bigger the working set size and the higher the throughput, the more time will be spent on GCs. Besides, execution efficiency at the CPU will typically be hurt when more elements are referenced due to an increased probability of cache misses. This is particularly the case when events are represented as Objects, because there is no guarantee that consecutive elements in the window will be allocated contiguously in the heap by the JVM. Therefore, even though the algorithm logically traverses the window in a sequential way, the memory access pattern tends to be essentially random. One possible way of eliminating this undesirable effect is to employ the column-oriented storage model, which avoids the random walks through the heap by keeping attributes as arrays of primitive types. However, this approach is worthwhile only if the number of attributes referenced in the query is small. Otherwise, consuming an event from the stream and inserting it into the window will involve accessing several memory locations (one entry per attribute in different arrays).

We tested, then, a tuned algorithm to minimize this performance penalty due to multiple inserts. The idea is to avoid references to distant memory locations by using a L2-resident temporary buffer for accommodating the incoming events. This temporary buffer consists in N aligned arrays (one per attribute) as in the original window, but with a capacity of only 100 events. Once these small arrays get full, the events are copied back to the window, one attribute at a time, so that they can be expired later.

The algorithm is described in detail in Table 3. Figure 8 compares the performance of the proposed algorithm with the original column-oriented implementation.

Table 3: Cache-Aware Algorithm for Computing Aggregations over Sliding Windows

```
Input:   S: incoming event stream
         K: the size of the L2-resident temporary buffer
Output: R: stream of results

for each event $E$ in $S$ do
    for each attribute $A_i$ of event $E$ do
        store $A_i$ on the corresponding temporary location $T_i$
    compute aggregation (update aggregator state)
    insert aggregation result into output stream $R$
    if temporary buffer $T$ is full then
        for each attribute $A_i$ of event $E$ do
            for each item $I_j$ in temporary buffer $T_i$ do
                copy $I_j$ to the appropriate location in corresponding window $W_i$
        reset the temporary location $T$
        slide the window $W$ in K positions
```

The optimized algorithm provided gains in performance that ranged from 28 to 35 percent when compared to the original column-store. The hardware metrics confirmed that it indeed exploits better the characteristics of the CPU: the CPI was almost half of the CPI of the original column-store and L2 cache miss rate was reduced to around 70% of what was originally measured. Evidently, this microarchitecture-aware algorithm is best-suited for medium-to-large windows, since for smaller sizes the working set of the original column-oriented implementation already fits in L2 cache.

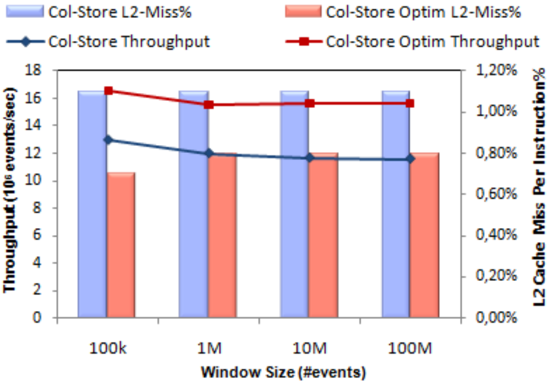


Figure 8: Performance Comparison: Conventional Column-Store Algorithm vs. Microarchitecture-Aware Algorithm

Join Query

In theory, the cost of a lookup on a hash table should be independent on the number of elements stored on it, but this ideal rarely can be achieved in practice since CPU operation – specifically cache behavior – is considerably affected by working set size. Figure 9 illustrates the correlation between microarchitectural aspects and final system performance for the join tests with tuples represented as POJO.

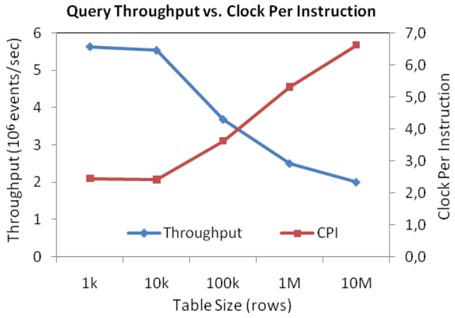


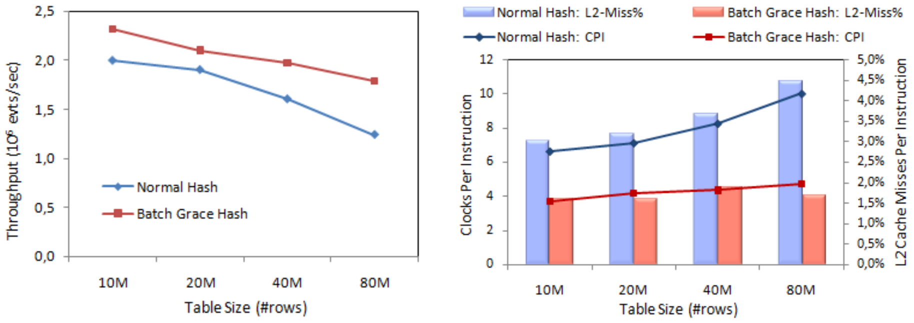
Figure 9: Drop on Join Performance due to Increase in CPI

As it can be noticed from the graph, query throughput falls in the same proportion as the clock per instruction metric increases, indicating that microarchitectural factors were the major cause for performance degradation when the table size was increased. Further analysis of the hardware metrics confirmed that the increase in CPI was due to less efficient memory access patterns. For example, as the table size was increased from 1,000 rows to 10M rows, *L2 cache miss per instruction* metric went from 0.6% to 3.1% and *TLB miss penalty* metric jumped from 0.3% to 19.4%. Notice also that up to 10k rows, the table fits in the 12MB L2 cache, which explains the negligible performance drop from 1k to 10k, and the significant degradation from that point on.

To improve data locality, we implemented an adapted version of the grace hash join algorithm [9] used in DBMSs. The idea is to reduce the number of times data is brought from main memory to cache by splitting the whole table into partitions and accessing them in bulks. The algorithm works as follows:

- When the table is being populated, the records are stored into partitions using a given partitioning function *g* (in our tests the table was split into 1000 partitions);
- Incoming events are then processed in batches. They are buffered into a partitioned list until the batch is complete. (The partitioning function is the same as the one used for splitting the table, which ensures that matching tuples in the batch and the table will be kept in corresponding partitions).
- Once the batch is complete, the corresponding partitions from event batch and table are loaded in pairs. The event partitions are then sequentially scanned, performing for every event a lookup on the corresponding table partition.

Figure 10 shows test results with both the conventional hash join algorithm and the batch grace hash algorithm, for table sizes ranging from 10M rows to 80M rows.



(a) (b)

Figure 10: Conventional Hash Join vs. Batch Grace Hash Join

As it can be seen in (b), the batch algorithm was successful in improving locality of data accesses, which in turn caused a reduction in average CPI. This resulted in performance gains that ranged from 11 to 44 percent, as illustrated in (a).

Notice that there are a couple of competing factors influencing the performance of the batch grace hash algorithm. For example, each table partition should ideally fit in L2 cache in order to minimize the high penalties associated with memory accesses. Assuming that table size is application-specific and cannot be changed, the only way this can be achieved is by increasing the number of partitions in the table. Doing so, however, means that the number of partitions in the batch of events is also increased, thus reducing the number of events per partition. A reduced number of events per partition will probably hurt performance as a good fraction of the lookups in the table will incur in compulsive cache misses. For avoiding this to happen, batch size could be increased in the same proportion as the number of partitions, but obviously this is only feasible if there is availability of memory resources. Determining the optimal values for these parameters is subject for further investigation.

3.6 Optimizations in a Multi-Query Scenario

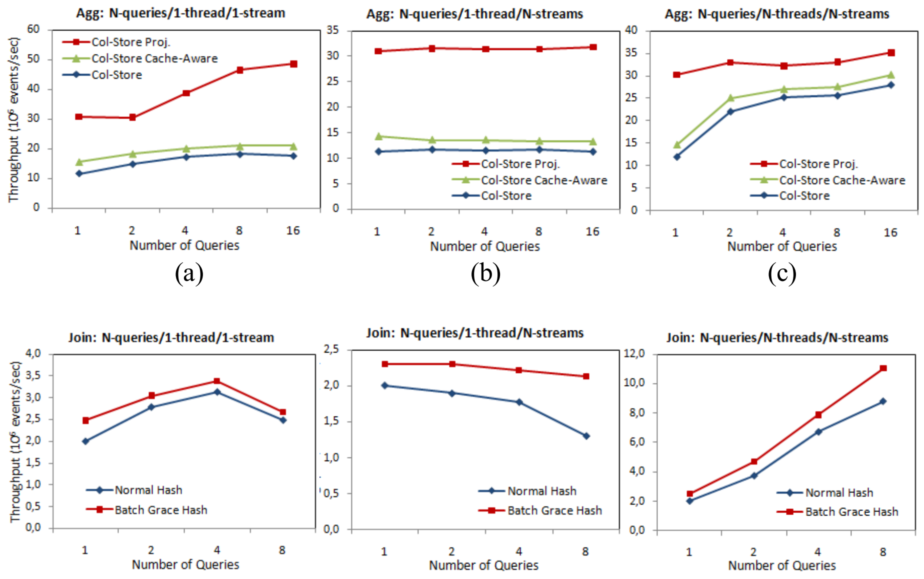
A natural question that might arise after analyzing the results of previous Sections is whether similar improvements in performance would be observed when moving from a scenario with only one continuous query running at a time to a multi-query scenario. In this Section we answer this question and present the results for a set of tests in which the proposed optimizations are validated by measuring system performance during the execution of multiple simultaneous queries. More specifically, we tested three different situations:

i. N instances of the same query are computed over a *single event stream* in a *single thread*;
ii. N instances of the same query are computed over *independent but identical event streams* in a *single thread*;
iii. N instances of the same query are computed over *independent but identical event streams* in *N threads*.

For aggregation, we performed tests with 1 up to 16 simultaneous queries, with sliding windows of 10 million events each. For join queries, we tested 1 up to 8 simultaneous queries operating over N tables of 10 million records (there was no physical memory available to test with 16 queries). We then analyzed the evolution of throughput and hardware-level metrics as we progressively added more queries to the configuration. The output throughput of each setting is shown in Figure 11.

Application-level and hardware-level metrics collected during tests indicate that the proposed optimizations are also effective in a multiquery scenario. For instance, the microachitecture-aware aggregation algorithm introduced in Section 3.5 achieved superior performance than the conventional column-store in all multi-query tests. Also, the "*Col-Store Proj*", who achieved the highest throughputs in the single aggregation query scenario due to improved microarchiecutral performance was once more the best performing implementation.

For join, the speedup of the batch grace hash algorithm over the conventional hash algorithm oscillated between 1.02 to 1.64. On average, the optimized implementation achieved 22 percent more throughput than the conventional non-optimized one, a slightly better speedup than the one observed in the single query scenario (20 percent). Once more, hardware-level metrics followed the trend observed in the single query scenario (on average, the conventional hash algorithm had a CPI of 7.52 and L2 cache miss rate of 3,5% against a CPI of 3.46 and a L2 cache miss rate of 1,6% for the batch grace hash algorithm).



(a) (b) (c)

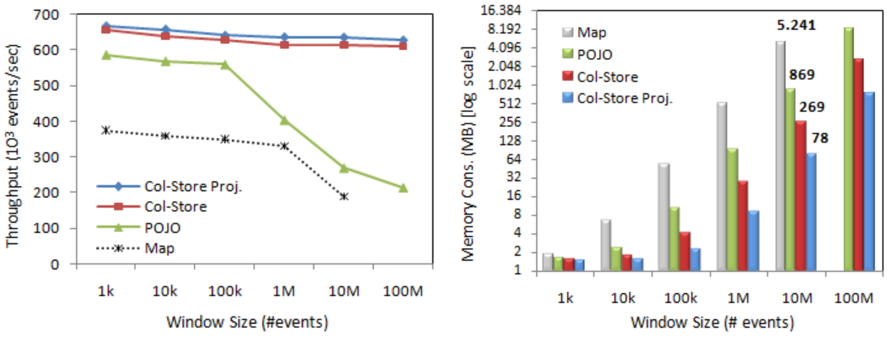
(d) (e) (f)

Figure 11: Output Throughput for Aggregation and Join Queries in Multi-Query Scenario

A few words about the shape of the curves: on (a) the output throughput incresased sligthly when more queries were added as a result of a reduction on the relative weight of event instantiation on the workload (event is created once, but processed N times). This constrasts with the workload on (b) where event instantiation is replicated in the same proportion as queries, which explains the steady line (individual throughput of queries decreases, but total system throughput remains the same). Interestingly, the curves on (c) did not present a linear (*or close to linear*) growth that one would expect when increasing the amount of resources for a set of independent tasks (as happened on (f), for example). We found out that this happened because the CPI metric in these tests increased essentially in the same proportion as the number of queries. The reason for this behavior, however, is unclear to us since the other hardware metrics (i.e., cache miss rates, instruction fetch stalls, resource stalls, etc.) did not show any change that could justify this increase in CPI. Finally, it should be noticed that the drop on performance when jumping from 4 to 8 queries on (d) was caused by increased garbage collection activity (for 8 queries the system ran close to the maximum memory available).

3.7 Optimizations on a Real EP System

In this Section we discuss the results of the tests performed with the event processing engine Esper, after implementing the proposed optimizations. Figure 12 below shows the most relevant findings.



(a) (b)

Figure 12: Optimizations on Esper (Aggregation Query):
(a) Throughput (b) Memory Consumption

As in the tests with the EP prototype, the modified version of Esper using a column-oriented storage model achieved higher throughputs and scaled better than the original implementations using Maps or POJOs as event representation. The column-oriented implementations also proved once more to be useful for reducing memory consumption for aggregation queries.

It should be noticed however, that many of the performance gains obtained with other optimizations in the tests with the prototype were hidden by bottlenecks inherent to Esper implementation. This phenomenon can be seen in Figure 12-a, where the "*Col-Store Proj.*" implementation achieved only 2,7% more throughput than the conventional "*Col-Store*", while on the EP prototype the same optimization provided twice more performance. For the same reason, the cache-aware algorithm for aggregation and the batch grace hash join algorithm provided only slight performance gains on Esper (around 1 percent).

4 Conclusions and Future Work

In this paper we started the discussion on how event processing systems can execute more efficiently at processors of today. We first collected application and hardware-level metrics during execution of a real event processing system to determine its performance at CPU. After identifying some issues, we proposed, implemented and evaluated some changes in data organization and algorithms that together provided more than order of magnitude performance gains and considerable reductions on memory consumption. We have also found out that:

1. Microarchitectural aspects played a fundamental role on the performance reduction observed when input size was increased;
2. The column-oriented storage model can greatly improve performance for aggregation queries when compared to alternative representations like Plain Java Objects or key-value Maps. These performance gains, whose roots are in an improved microarchitectural execution as well as in a reduced garbage collection activity, are also followed by significant memory savings (56% up to 98% reduction);
3. The proposed cache-aware optimizations remained effective in a scenario with multiple queries running simultaneously.

As future directions, we plan focus in the development of more optimized structures for join queries as well as to work in the optimization of other classes of continuous queries (e.g., event pattern detection). Other interesting topics worth of further investigation include how to make a better use of the parallelism provided by modern multicore processors and the compromise between CPU-level optimizations and query-plan sharing.

References

1. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In Proc. of the 2008 ACM SIGMOD Vancouver, Canada.
2. Ailamaki, A., DeWitt, D.J., Hill, M.D. Wood, D.A.: DBMSs On A Modern Processor: Where Does Time Go? In Proc. of the 25th VLDB Conference, Edinburgh, Scotland, 1999.
3. Ailamaki, A., DeWitt, D.J., Hill, M.D. Wood, Skounakis, M.: Weaving Relations for Cache Performance. In Proc.of the 27th VLDB Conference, Roma, Italy, 2001.
4. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB Journal, Vol. 15, Issue 2: 121-142 (2006).
5. Arasu, A. Widom, J.: Resource Sharing in Continuous Sliding-Window Aggregates. In Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004.
6. Esper, http://esper.codehaus.org/
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, 4th edition, 2007.
8. Intel VTune: http://software.intel.com/en-us/intel-vtune/
9. Kitsuregawa, M., Tanaka, H., Moto-Oka, T.: Application of Hash to Data Base Machine and Its Architecture. New Generation Computing 1 (1): 63–74, March 1983.
10. Mendes, M.R.N., Bizarro, P., Marques, P.: A Performance Study of Event Processing Systems. In Proc. of First TPC Technology Conference, TPCTC, Lyon, France, 2009.
11. Ravishankar Ramamurthy, David J. DeWitt. Buffer-pool Aware Query Optimization. CIDR 2005: 250-261
12. Stonebraker, M., Abadi, D.J., et al: C-Store: A Column-Oriented DBMS. In Proc. of the 31st VLDB Conference, Trondheim, Norway, 2005.
13. Zukowski, M., Boncz, P.A., Nes, N., Heman, S.: MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Engineering Bulletin, Vol. 28, No. 2: 17-22 (June 2005).
14. Zukowski, M., Heman, S., Nes, N., Boncz, P.A.: Super-Scalar RAM-CPU Cache Compression. In Proc. of the 22nd ICDE, Atlanta, USA, 2006.