A Cost Model for Data Stream Processing on Modern Hardware

Constantin Pohl
TU Ilmenau, Germany
constantin.pohl@tu-ilmenau.de

Philipp Götze
TU Ilmenau, Germany
philipp.goetze@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

For stream processing application domains, using queries to process or analyze data incoming from potentially endless streams, low latency and high throughput are key requirements. It is not easy to achieve this as many factors influence the actual runtime of query execution plans and one can not measure all of them individually. Therefore, query optimizers try to overcome this hurdle by using cost models for decision making. Modern hardware architectures and devices, like manycore CPUs or the NVRAM storage technology demonstrate new properties for query execution, which have not received much attention within the model. Thus, traditional optimizers are not capable of dealing with these new factors leading to results possibly far away from optimum.

Our work addresses this problem providing a new cost model based on modern hardware characteristics. We analyze hardware aspects necessary for query optimization and substantiate them with our own low-latency stream processing engine PipeFabric. This yields in a cost model that can precisely predict the performance of query execution plans on modern hardware close to actual measurements.

Keywords

Cost Model, Xeon Phi, KNL, MIC, NVRAM, Stream Processing

1. INTRODUCTION

Nowadays, many applications in different domains ranging from monitoring cyber-physical systems over IoT to finance and healthcare require the processing of data streams often with low latency. This is typically addressed by stream processing engines (SPE) pioneered by systems like STREAM [2], Aurora [1] or TelegraphCQ [14] and today by massively parallel engines such as Apache Storm [16] or Flink [4]. Apart from efficient algorithms for – possibly incremental – processing and implementation techniques, query planning and optimization play an important role.

Proceedings of the VLDB Endowment, Vol. 10, No. 9
Copyright 2017 VLDB Endowment 2150-8097/17/05.

Achieving low latency requires utilizing the available hardware in the most efficient way. Examples are manycore architectures such as Intel's Xeon Phi series allowing a high degree of parallelism, co-processors such as FPGAs and GPUs for offloading compute-intensive tasks but requiring data transfer between host and co-processor, SIMD instruction sets such as AVX512 for efficient processing of small batches of data, cache-optimized implementations utilizing processor caches or high-speed networking supporting new communication models such as RDMA. However, such aspects are taken into account to a very limited extent in current cost models for stream processing preventing query optimizers to explore the plan space also in terms of hardware parameters, e.g. the number of threads (determining for example the number of partitions) used for a given query, vectorization of processing as trade-off with latency, using different storage layers (memory technologies) for managing the state of operators and much more. Another open question is the required level of detail of such a cost model to allow to choose the optimal plan for different hardware parameters.

Thus, the goal of our work is to empower query optimizers for stream engines to take hardware features into account by proposing a hardware-oriented cost model. Particularly, we consider manycore architectures exemplified by Intel's Knights Landing (KNL) architecture and the problems of stream partitioning/degree of parallelism as well as operator state management with different memory technologies. Our contribution is twofold:

- We present a cost model for parallel stream processing based on the idea of a rate-based model that specifically considers hardware properties of a manycore processor architecture with different memory types.
- We report results of an experimental evaluation showing the viability of the proposed model.

2. RELATED WORK

For optimizing queries, [11] shows that todays optimizers are often very complex without providing much better results than simpler approaches. Large estimation errors occur frequently, leading to bad execution plans even on relational databases without streaming purposes. Machine learning techniques are stated as modern solutions, outstripping traditional cost model practices. However, [6] challenges this statement, showing that a well calibrated optimizer using a cost model delivers excellent results.

For hardware-oriented cost models, [13] provides deep insights on hardware factors used in database queries. They analyze the performance of queries with the help of a cost model for algorithms used by database operators as well as the influence of hardware properties, especially CPU and memory costs. Nevertheless, stream processing has its own features (further explained in Section 3) where cost models of traditional relational databases are not applicable.

In [9] they address these stream processing characteristics for query optimization. They cover the entire range from streaming properties to query execution, providing usable cost models for the optimizer considering various operations and additionally describing how to dynamically integrate optimized plans into already running database systems. While they are a solid base, for streaming applications running on modern hardware they are too inaccurate, lacking partitioning strategies for better query performance among others.

The partitioning aspect is explicitly addressed by [12] on intra-query parallelism. When applying multithreading with partitioning, it is necessary to decide where partitioning can provide performance improvements. They use heuristics on selectivity and costs of operators as well as on table size and query plan structure. Concerning stateful stream processing, [5] has examined various partitioning functions in great detail.

To bridge to specific hardware, there is already work done on co-processors like GPUs from [3] or [7]. The main focus on work with co-processors lies on overcoming the bottleneck of data transfer between host and GPU, which also applies for the predecessor of the KNL manycore CPU, the Xeon Phi Knights Corner.

3. DATA STREAM PROCESSING

Many modern applications require stream processing since the data would be too huge to store or is only valid within a specific time frame. The data typically arrives continuously represented by tuples, possibly endless and with fluctuating rates. To deal with scopes or dependencies among the tuples, window semantics are usually provided.

Handling high data rates to achieve the often purposed real-time characteristics requires a partitioning of streams and states as well as the parallelization of the query operators. The remainder of this section, therefore, describes our own SPE *PipeFabric*[1] and how parallelism and partitioning are realized inside. In addition, we also consider stateful operators for the cost model, where it is necessary to access (possibly persistent) external storage regions.

3.1 Stream Processing Engine (SPE)

In an SPE queries are often represented and internally organized as dataflow graphs or pipelines. Compared with traditional engines such as STREAM [2] and Aurora [1], modern SPEs do not provide declarative languages like CQL but language integrated APIs or domain-specific languages (DSLs). This makes it possible to express more complex queries, which can easily support application-specific functionalities in the form of user-defined functions through the underlying programming language.

With our internal SPE *PipeFabric* written in C++, queries can also be formulated using a DSL by putting together sequences of operators to construct a dataflow. Besides various standard operators such as projection, filter, join, and aggregate, it also supports complex event processing and tables to combine the worlds of stream and batch processing.

3.2 Parallelism

There are basically two strategies to parallelize queries, namely inter- and intra-operator parallelism. Inter-operator parallelism can be achieved by separating the dataflow into multiple parts which can run in parallel on different threads. For intra-operator parallelism, multiple instances of an operator are created to partition the load.

In *PipeFabric* both forms of parallelism are realized, shown in Figure 1. With synchronized queues for tuple exchange between operators, parts of the query can be decoupled and run simultaneously with multiple threads, providing inter-operator parallelism. Intra-operator parallelism uses a partition-merge schema. Here, each partition is decoupled in its own thread executing a specific operator (sequence). Additionally, a partitioning operator is preceded that takes care of splitting the incoming stream and forwards the tuples to the matching partition. At the end, a merging operator is responsible for combining the results of all instances into a single stream again.

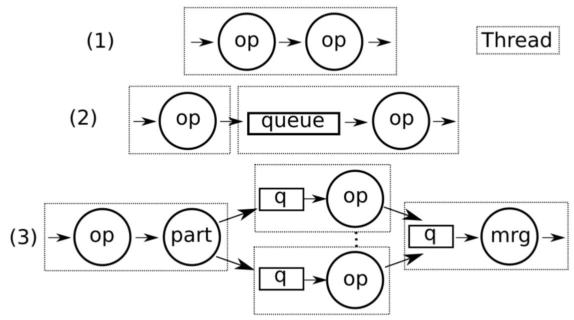

Figure 1: Multithreading in a Query
(1) Singlethreaded execution
(2) Multithreading between operators
(3) Partitioning-Merge schema

However, utilizing parallelism does not always lead to better performance. The partition and merge step as well as synchronization efforts for exchanging tuples between threads are causing a notable overhead. A singlethreaded operator, on the other hand, can only process a limited number of tuples in a certain time frame. Thus, it is a tradeoff where an optimizer has to decide when it is useful to parallelize using a well-defined cost model.

3.3 Stateful Operations

Some streaming operators require to hold and access stored items such as metadata, history data, intermediate results, etc. Typical operations are for example windows, aggregations, and joins. We subsume this kind of operators as stateful operations. In contrast to stateless operations, which can process each tuple individually and independent of previous tuples or other data items, stateful operations require additional accesses to memory and storage. Considering a singlethreaded streaming application, especially the access granularity (byte or block) and the read-write ratio of an algorithm is determined by the used hardware. That is, depending on the hardware one might choose an algorithm using more intermediate writes or less writes with more reads

[1] https://github.com/dbis-ilm/pipefabric

and recomputations instead (cf. Section 4). When utilizing parallelism for stateful operations these considerations are getting more complex. One has to take partitioning, distribution, synchronization, and others into account and pick the correct strategies to satisfy the characteristics of the underlying medium. In a cost model preferably all runtime components of reading/writing to the storage should be considered such as indexing, concurrent access, CPU caches, OS cache, etc.

3.4 Query Optimization

Basically, query optimization in database systems is a well-studied problem and a variety of methods and techniques have been proposed to support cost-based decisions about operator ordering and algorithm selection, but also for scheduling strategies at runtime. For the specific case of stream processing dedicated cost models are required differing from standard database cost models because the underlying idea of cardinality estimation does not make much sense for the processing of possibly infinite streams. One example of an appropriate cost model is the rate-based model from Viglas and Naughton [17] which uses operator input and output rates instead of cardinalities. Since a stream is running potentially endless and a query can produce results with fluctuating rates, an optimizer has to check if the query still runs in the most effective way or may need to be dynamically adapted. To point an example, if some keys of incoming tuples appear more and more frequently and the query executes a hash join on each of them based on a huge internal hash table, it would be a good idea to pull the frequently requested hash partner into the cache. However, such a full optimizer is out of the scope of this paper and will be further investigated in later work.

Because an optimizer needs a cost model to base its decisions on, we want to provide such a model for modern hardware usage, especially focusing on multithreaded query execution and special memory systems.

4. HARDWARE PROPERTIES

Modern hardware is a wide topic. Within the scope of this work, we focus on CPUs and memory, because they determine most of the performance and query costs. In specific manycore processors and NVRAM have been examined more deeply and are briefly introduced here.

4.1 Manycore Architecture

A manycore processor uses much more cores than a conventional multicore CPU, leading to its name. While Nvidia focuses on GPU technology, Intel switches to general manycore architectures with the Xeon Phi series. The latest CPU in this series, the KNL, has up to 72 cores. Each of the cores supports four threads, available through Intel's hyperthreading technology. This leads to a massive degree of possible parallelism. However, so many cores on a small chip create an intense waste heat that has to be dissipated. This problem is solved by simpler cores and a lower clock speed – each core of the KNL utilizes only up to 1.50 GHz. The result of this low clock speed is a significantly poorer performance when running an application singlethreaded.

To overcome this problem, the KNL provides in addition to its parallelism the next level of instruction sets, called AVX512. With this set, the register width is doubled, allowing a better SIMD speedup than traditional CPUs. Another improvement is the multi-channeled DRAM on chip (MCDRAM) with up to 16GB size and a supported bandwidth of around 320GB/s. With many threads running in parallel, the MCDRAM allows them to fetch data from memory without overburdening the memory controllers, leading to generally the same latency even with lots of memory requests.

The most promising aspect of a manycore architecture is the possible parallelization degree. In the scope of this paper, our model, therefore, focuses initially on exploiting that parallelization with multithreading, especially partitioning and decoupling operators.

4.2 Non-Volatile RAM

The traditional way of storing data has always dealt with the performance gap between DRAM and disk. To access and recover the data even after a power loss, it is necessary to write the data to a persistent medium. These devices, however, are typically only block-addressable and work faster with sequential access. In contrast, DRAM can be accessed randomly with byte granularity, but the data is volatile.

With NVRAM one tries to merge both technologies and their advantages into one device. The most promising representatives are phase-change memory (PCM) [10], SST-MRAM [8] and memristors [15]. They provide fast read-write latencies and byte-addressability similar to DRAM. NVRAM also adds direct persistence and small memory cell density like SSDs or HDDs allowing for higher capacities. However, there are also undesirable aspects like the read-write asymmetry and limited cell endurance in some technologies (e.g. PCM). For the optimizer, this means that besides the latency considerations for the cost model, also the algorithms must be chosen with regard to these characteristics.

5. COST MODEL

An optimizer improves the execution of a query for lower latency or more throughput. While latency describes (approximately) how long it takes for a tuple to be fully processed by a query, increasing throughput allows more tuples to be processed in a certain time frame. For real-time critical applications, it is necessary to produce a result as fast as possible, a property that manycore processors do not fulfill compared to multicore CPUs, caused by lower clock speed resulting in poor singlethreaded overall performance. However, throughput can theoretically be increased by a tremendous amount when utilizing as many cores as possible.

With multithreading in mind, the cost model is explained later in this section. First, relevant factors of hardware are mentioned. Next, requirements on stream processing are discussed. Finally, they are combined along with further explanations.

5.1 Hardware Factors

Hardware properties strongly influence optimization goals and performance gains. To point an example, the KNL 7210 supports up to 256 threads with only 1.30 GHz clock frequency per core. For comparable throughput with equal tuple arrival rates from source, it has to use more threads sooner (in terms of operator complexity) as a multicore CPU with high clock frequency.

Measurements on hardware are tricky, though. Modern processors use lots of techniques for hiding latencies and memory accesses to massively increase performance. This makes it difficult to get reliable measurements without influence from other sides. The hardware prefetch mechanism, for example, is one of those techniques, pulling data from memory into caches in advance, concealing real memory access times.

Relevant hardware factors can be split into three main categories, CPU, main memory and caches, as shown below. State-related factors and costs are regarded separately later.

CPU. For CPU, we regard two main factors for performance. The **clock frequency** determines how fast any instruction is executed. A lower clock speed means that it takes more time for a processor until each instruction is performed, worsening latency in general. The **number of supported threads** derives the possible effective degree of parallelization with multithreading.

Main Memory. On main memory, its **size** is responsible for the amount of data that can be used before accessing the disk. Today, main memory is not very expensive anymore, allowing sizes of hundreds of Gigabytes easily. The **latency of main memory access** determines how fast data can be pulled into caches to make it ready for processing by the CPU.

Caches. Caches speed up runtime of applications by providing extremely fast data access when the requested data is already present in a cache. It is always a tradeoff between the size of a cache and the latency of accessing elements inside. So most of the time each core has its own small cache with extremely low latency for data (L1d) and instructions (L1i) as well as shared lower level caches between cores that are bigger with higher latencies (L2, L3). Relevant factors for a cost model are how many caches a system has, with certain attributes on each cache. The **size** of the cache along with its line size determines how many elements it can effectively hold. The **latency of cache access** can be fine-grained by distinguishing between hits and misses, on sequential or random access. We use the general hit latencies in our case.

Table 1 summarizes the relevant hardware factors discussed above.

Hardware Factor	Symbol
clock frequency	f_{clock}
number of threads	num_{thread}
main memory size	mem_{size}
memory access latency	mem_{lat}
size of cache j	Lj_{size}
hit latency of cache j	Lj_{lat}

Table 1: Relevant Hardware Factors

5.2 Stream Processing Model

An SPE supports different operators that can be applied combined in a query on a data stream. How these operators are implemented differs in each engine. For example, a join between two streams can be performed by a hash or a sort-merge join, each with different costs of memory usage, CPU processing time and cache reuse. Our focus in this work is on the multithreading aspect with its impact on performance based on underlying hardware. However, up to a certain degree, a fine-grained consideration of operator costs is necessary.

The cost of a query represents the amount of work done before it returns a result. For stream processing purposes, our goal is to minimize the approximated latency per tuple $lat(tp)$. A better throughput results into more processed tuples per time frame, what can also be expressed in $lat(tp)$ by lower processing time per tuple. The overall formula is given in Equation 1, where c_q denotes cost of a query and tp_{proc} is the number of processed tuples in a certain time frame.

$$lat(tp) = \frac{c_q}{tp_{proc}} \tag{1}$$

The singlethreaded query costs c_{qs} simply consist out of the sum over all operators it uses including their selectivities. When a tuple arrives from source, n operators are applied sequentially on that tuple when running singlethreaded. Each operator modifies it potentially and forwards it to its successor. On singlethreaded execution, the transfer costs between operators are negligible and therefore not of further interest in the equation. However, the efforts of an operator mainly depend on how many tuples it has to process, denoted with the parameter tp_{in}. This is expressed in Equation 2.

$$c_{qs} = \sum_{i=1}^{n} (c_{op(i)} \cdot tp_{in}) \tag{2}$$

The usage of multiple threads on a single query can be realized in two ways, according to subsection 3.2. First, we show the costs for a query with inter-operator parallelism (see Equation 3).

$$\begin{aligned} c_{qm} = max(\sum_{i=1}^{k} (c_{op(i)} \cdot tp_{in}), \sum_{i=k+1}^{n} (c_{op(i)} \cdot tp_{in})) \\ + c_{queue} \cdot tp_{in} \end{aligned} \tag{3}$$

The c_{queue} costs describe the delay for exchanging tuples between the threads. It depends on the synchronization mechanism that is used by the SPE, e.g. locks or timestamps. Because of simultaneous execution of threads, the slower thread determines overall costs in addition to synchronization efforts, expressed by the maximum in the formula. $k+1$ is the position after which the second thread takes over.

Next, we list the costs for a query with partitioning-merge schema (shown in Equation 4).

$$\begin{aligned} c_{q\_multi} = max(\sum_{i=1}^{k-1} (c_{op(i)} \cdot tp_{in}) + c_{part} \cdot tp_{in}, \\ (c_{queue} + c_{op(k)}) \cdot tp_{in}, \\ (c_{queue} + c_{merge}) \cdot tp_{in} + \sum_{i=k+1}^{n} (c_{op(i)} \cdot tp_{in})) \end{aligned} \tag{4}$$

The outer maximum determines overall costs by three components. The first component is the thread running all preceding operators plus the partitioner. The second one is the partitioned operator, multiplied with incoming tuples. The last component is the thread running the merge step with all following operators. With more partitions, less tuples need to be processed by each of them. But if the bottleneck is on the first or the third component, performance is not improved by adding more partitions.

5.3 Stateful Operations

As described earlier, we also pursue to integrate the storage part of the hardware in our cost model as it is crucial in particular for stateful operations. For that, it mainly depends on the type of device holding the state such as magnetic disks, SSDs, NVRAM, or volatile memory only. Even for NVRAM, it can make a big difference which technology is used, i.e. PCM, SST-MRAM, memristor, or whatever may come in the future.

In itself insignificant for the cost model but important for the algorithm is the access mechanism. Hence, one has to use the correct interfaces such as filesystems, allocators, or direct access to store and retrieve the data correctly. The relevant part for our cost model is then the resulting latency for accessing the state. Thereby, it is necessary to distinguish between read and write latency (denoted S_{r_lat} and S_{w_lat}) due to the read-write asymmetry for most of the storage technologies. In addition to the advertised performance characteristics of the single device, it is important to measure the real values within the used system by small experiments or benchmarks inspired by the intended workload. These values can greatly differ through various caching mechanisms, NUMA effects, hardware combinations etc.

On top of that, one also has to take logical costs into account similar to Manegold [13]. Typical factors here are the data volume and the selectivity of the state predicate. In this scenario, the average number of read and write operations per tuple seems to be the factor of interest, which can often be derived from the data cardinality and the type of state. For example, using partitioned hash-join with a static data set should only require one lookup in the state per tuple (possibly cached for each partition). Considering a grouping or grouped aggregation may require at least one write and multiple read operations depending on the size and organization of the state. Thus, the number of reads and writes is mainly determined by the operator and shape of the state. First experiments have shown us that simply estimating the number of reads and writes does not give the correct cost due to various caching and optimization effects. That is why an operator-dependent state manipulation factor for reading and writing (denoted $f_{<op>\_Sr}$ and $f_{<op>\_Sw}$) needs to be calibrated. When dealing with custom states behaving like a black box, things get more complicated and one has to fall back on heuristics, statistics, or annotations.

Combining the logical and physical costs one can derive the total estimated execution time. This results in Equation 5 describing the access costs for a partitioned stateful operator per tuple.

$$c_{S\_<op>} = S_{r\_lat} \cdot f_{<op>\_Sr} + S_{w\_lat} \cdot f_{<op>\_Sw} \tag{5}$$

Another important factor is also whether a page cache by the operating system is interposed or the quite new DAX code is enabled to provide direct access to byte-addressable devices. In this way, unnecessary copies within the main memory will be avoided. If the medium is exclusively block-addressable it is important, on the one hand, to keep cohesive state data on the same block to reduce the number of external operations. On the other hand, for the optimizer, this could mean that partitions should be created block-based instead of tuple-based. Hence, one can drive various state partitioning strategies depending on the underlying medium. For the cost model, we need strategy dependent factors similar to the operator costs resulting in Equation 5 using these factors instead (denoted c_{S_part}). As expressed by the formulas earlier, partitioning is worthwhile only if the synchronization effort for partitioning and merging the state plus the longest running thread is smaller than for a singlethreaded execution. It is expedient to organize the state in a way enabling a disjunctive division of the data to avoid merge synchronization and general communication overhead among the threads. For fixed length entries in such states, this is quite easy to achieve but may increase the lookup time for big states.

Another strategy, especially for smaller states, is to partition only the incoming tuples and not the state. This state-sharing technique would additionally require a measure for the synchronization overhead, which would further enhance the formula.

Besides the performance considerations, an optimizer and its cost model should also try to optimally utilize the specific characteristics of the hardware class or deal with its drawbacks. In the case of NVRAM, the read-write asymmetry and lower cell durability in comparison to DRAM require that optimizers use write-limited plans if possible. For block devices it is especially important to strive for sequential instead of random access to avoid additional seek times (e.g. arm movement) or block accesses. It is quickly clear that a lot of factors come into play depending on the hardware and also the type of the state. For the sake of simplicity, we stay with Equation 5 for the time being in this paper.

5.4 Hardware-Conscious Cost Model

In this section, we combine our hardware facts with the stream processing formulas, regarding operator costs c_{op}. These costs depend on optimization degree of the compiler, algorithm costs, implementation details and much more, which are out of the scope of this paper, simply noted as c_{cpu} costs in the following formulas.

Projection. The projection is an unary operation which restricts incoming tuples to certain attributes. In our current implementation, it gets a pointer on a tuple per subscription from the preceding operator or generator. For the projection function, it has to access the data behind the pointer, which should be cached ideally. However, its result has to be written again in main memory. Equation 6 shows the notation.

$$c_{proj} = L1_{lat} + mem_{lat} + \frac{c_{cpu}}{f_{clock}} \tag{6}$$

Selection. The selection operator drops tuples that do not fulfill a predicate. Therefore, it reads the necessary attributes from incoming tuples (from cache), evaluating the predicate and publishing the tuple for the next operator if the evaluation is true. No main memory is involved here, so this operator is very fast. Equation 7 shows the costs.

$$c_{sel} = L1_{lat} + \frac{c_{cpu}}{f_{clock}} \quad (7)$$

Aggregation. The aggregation operator applies a stateful aggregate on each incoming tuple. It has to access and update its state, reading from the cache and writing it to main memory as well as accessing the necessary attributes for aggregation, ideally cached also. The costs can be written as in Equation 8.

$$c_{aggr} = 2 \cdot L1_{lat} + mem_{lat} + \frac{c_{cpu} + c_{S\_aggr}}{f_{clock}} \quad (8)$$

Grouping. The grouping operator applies an aggregate on each tuple with respect to a key column. Thus, the operator is a bit slower than the aggregation explained before. Costs are listed in Equation 9.

$$c_{grp} = 2 \cdot L1_{lat} + mem_{lat} + \frac{c_{cpu} + c_{key} + c_{S\_grp}}{f_{clock}} \quad (9)$$

Queue. The queue is used for multithreading and allows tuples to be exchanged between threads. According to its synchronization mechanism, it can possibly add high latencies. The queue is coupled to another operator, for example, the projection operator. In our current realization, the queue writes the tuple that has to be exchanged into memory, expressed by mem_{lat}. Therefore, it acquires a lock, releasing it thereafter with notification of the other thread. This shows c_{sync}. Equation 10 shows the costs of the queuing mechanism.

$$c_{queue} = mem_{lat} + \frac{c_{sync} + c_{cpu}}{f_{clock}} \quad (10)$$

Partitioning and Merge. The two operators are used for intra-operator parallelism. The partitioner evaluates a predicate deciding which partition is responsible for the currently processed tuple. It then forwards the tuple into a queue according to the appropriate partition. The merger uses its own queue where results of the partitions are stored before it publishes them in a single stream downwards. It can be expressed like in Equation 11 and Equation 12. For the partitioning step, the arriving tuple is ideally already in the cache from the previous operator. The queuing costs are simply added and c_{cpu} describes the predicate evaluation costs. In the case of merging the c_{cpu} costs are only for publishing the tuples from the input queue, resulting in a single stream.

$$c_{part} = L1_{lat} + c_{queue} + \frac{c_{cpu}}{f_{clock}} \quad (11)$$

$$c_{merge} = c_{queue} + \frac{c_{cpu}}{f_{clock}} \quad (12)$$

To combine the partitioning and merge cost model (see Equation 4) with stateful operation costs (see Equation 5), we derived Equation 13 as a total cost formula for a multithreaded stateful query. The costs for accessing the state are considered for both the partitioning strategy and the partitioned operators. Costs for manipulating the state occur for every incoming tuple. Partitioning the state, on the contrary, is only necessary once in the beginning and in the case of dynamic repartitioning. The merger only retrieves the results, e.g. an aggregation, and thus do not have to manipulate the state itself.

$$\begin{aligned} c_{q\_mS} = max(&\sum_{i=1}^{k-1}(c_{op(i)} \cdot tp_{in}) + c_{part} \cdot tp_{in} + c_{S\_part}, \\ &(c_{queue} + c_{S\_op(k)} + c_{op(k)}) \cdot tp_{in}, \quad (13) \\ &(c_{queue} + c_{merge}) \cdot tp_{in} + \sum_{i=k+1}^{n} (c_{op(i)} \cdot tp_{in})) \end{aligned}$$

With this cost model in mind, we put our considerations to the proof in the following section.

6. EXPERIMENTS

Within this section, we intend to show with various experiments how greatly hardware factors and partitioning influence latencies of any operator to verify our cost model. For that, we use the Intel Xeon Phi Knights Landing 7210 manycore processor, supporting 64 cores, 256 threads with a clock frequency of 1.30GHz per core. After describing first hardware-related calibrations, we demonstrate general behavior on multithreading with decoupling of operators and partitioning. Subsequently, the tuple processing rate from implemented operators is shown. Finally, we compare these results with our cost model by running some example queries, measuring real latencies in comparison to the cost model outcome.

6.1 Hardware Calibration

Hardware factors are measured by our calibration tool, written in C++. Memory and cache sizes can be read from Linux sysinfo because the KNL runs on Linux operating system. The number of supported threads is simply revealed through the C++ thread class from the standard library. Measuring the latencies is more tricky, though. For main memory latency, an object is written to memory repeatedly for a million iterations. When measuring cache latencies, the prefetching mechanism needs to be disabled or bypassed first. This is granted by random access on elements of an array. That leads to unpredictable accesses and denies any latency hiding from the prefetcher. For each cache, we use an array that fits into that cache, but is too huge for previous smaller caches, leading to consequent cache misses there. Results are shown in Table 2.

The KNL has no L3 cache, however, the MCDRAM can be used as a last-level cache when configured in cache mode. This is done here, even when its latency is worse than on main memory. Because we measured latencies with a single thread, the main advantage of the MCDRAM (its high bandwidth) is not relevant here, easily explained through the fact that a single thread cannot even saturate that maximum possible bandwidth (around 320GB/s).

6.2 Multithreading Behaviour

To get a better picture of how partitioning influences latency in our SPE PipeFabric, we run a single mathematical operator in a query with increasing complexity. It only adds tuple data on a single variable with certain repetitions. In real queries and operators, this complexity can be expressed by the number of operators a query uses as well as simply the CPU processing time for each of them. With increased complexity, the advantage of partitioning can be seen in Figure 2.

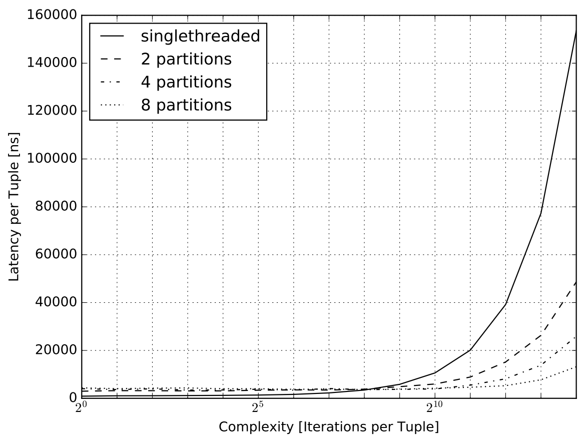

Figure 2: Partitioning Results

For low complexity, the synchronization overhead between partitions worsens approximated latency for each tuple. However, when CPU time increases, a notable speedup is achieved.

For decoupling operators into two threads, it is necessary to use it between two operators that benefit from simultaneous execution. If one operator is magnitudes slower than the other, it will just dominate the latency. Therefore we run the mathematical operator twice in the query, each of them in a single thread with synchronized data exchange. Results are shown in Figure 3.

With two threads in parallel, the approximated latency per tuple is finally close to a half compared to singlethreaded execution. For low complexities, however, the constant overhead by synchronization just worsens latency, same like on partitioning.

That means, a query optimizer regarding multithreading has to decide on a given query where it is most beneficial to add partitioning or decoupling, if at all.

6.3 Operator Measurements

Hardware Factor	KNL 7210
clock frequency	1.30 GHz
number of threads	256
main memory size	96GB
memory access latency	146.3ns
size of cache L1	32kB
size of cache L2	1MB
size of cache L3	16GB*
L1 access latency	3.1ns
L2 access latency	13.2ns
L3 access latency	172.7ns

Table 2: Measured Hardware Factors
*MCDRAM runs in cache mode

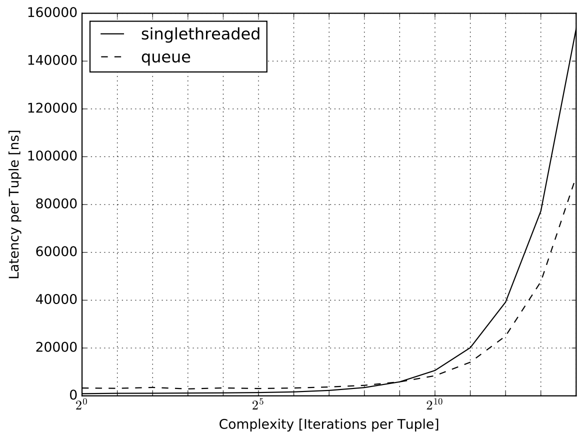

Figure 3: Decoupling Results

As already explained in detail in Section 5, we show the average latency each operator needs to fully process a tuple. We provide data through our generator, creating a million tuples sequentially with three attributes each, an integer, a double and a string. The integer is repeatedly counted up from zero to 10000, the double value is increased continuously beginning with 1.5 and the string contains six chars. After measuring its necessary runtime, we can run a query with each operator alone subtracting the generator time as constant. The projection operator reduces the attributes of incoming tuples to the integer value only. The selection operator has a selectivity of 20%, forwarding only one of five tuples. For aggregation, the operator sums up the double values from the tuples using an internal state. The grouping operator does the same with respect to the key of a tuple, represented by the integer value. Results are shown in Table 3.

Operation	Latency Costs
Projection c_{proj}	340ns
Selection c_{sel}	112ns
Aggregation c_{aggr}	552ns
Grouping c_{grp}	700ns
Queue c_{queue}	2650ns

Table 3: Measured Operations [Latency per Tuple]

With these measurements, we can split any costs into a memory access part and CPU processing part, according to our equations in Section 5.

6.4 Query Results

We use the following two queries for comparison between measured results and cost model outcomes:

- Q1: Selection (20% selectivity), Projection (integer and double attribute), Aggregate (double attribute)
- Q2: Projection (integer and double attribute), Grouping (double attribute)

For each query, it is run with a single thread (noted with Q s.), with decoupling by a queue (Q d.) and with partitioning

(Q p., two partitions). In addition, we use our cost model for an estimation, shown in Figure 4.

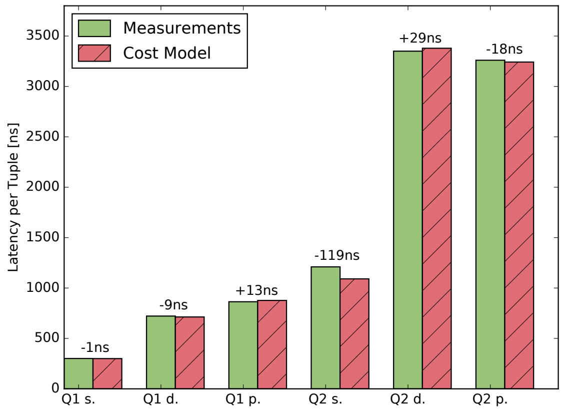

Figure 4: Query Results

As it can be seen, the cost model closely predicts the results measured directly when running a query.

We explain the numbers exemplarily for the first query, using the decoupling by a queue. On the top level, the costs of the query according to Equation 3 are the maximum of both threads plus the queue overhead. The first thread runs the selection, while the second thread runs the projection and aggregation. It can easily be seen that the maximum of both threads is the second one. Because of the selection operator returning just a fifth of tuples, the maximum is expressed as the sum of projection (Equation 6), aggregation (Equation 8) and queue (Equation 10) per tuple.

Relevant hardware factors like the clock frequency and cache and main memory access latencies are measured by our calibration step described above. The real CPU costs of an operator are tested outside of our PipeFabric framework with C++ tests written by oneself.

If we add all together, our cost model returns an approximated latency of 713ns per tuple, compared to real-time measurements on the first query that gives us 722ns.

Our model can be extended in future work for more exact results and better predictability on different hardware sets. With implemented locks realizing synchronization, the operators of the example queries are simply too weak in computational efforts for compensating that expensive synchronization. We will further enhance that mechanism in the future to provide a speedup with multithreading even for simpler operations.

7. CONCLUSION

In this paper, we addressed a part of the problem of query optimization on data stream processing using modern hardware. Decisions made by an optimizer are based on a certain cost model and statistics. We, therefore, provided a calibration approach to get important hardware factors like memory latencies or clock speed. On SPE side, we measured with experiments our algorithms and which hardware factors are necessary for the operations. With those results, we derived a cost model for stream processing queries to decide the ideal degree of multithreading. Even if simple operators and small queries do not benefit from parallelization, especially with expensive synchronization between them, this is recognized by our model.

It is possible to measure more important hardware influences, like the Translation Lookaside Buffer (TLB) or extending queries in such a way that most computation exceeds cache sizes. However, to get a solid foundation, we started at a higher logical level and integrated obvious hardware aspects first. It is expected that the results get more exactly when additional influences are considered, particularly when using different hardware sets on CPU and memory.

8. REFERENCES

[1] D. J. Abadi, D. Carney, et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB*, pages 120–139, 2003.
[2] A. Arasu, B. Babcock, et al. STREAM: The Stanford Stream Data Manager. Technical report, 2004.
[3] S. Breß, H. Funke, and J. Teubner. Robust Query Processing in Co-Processor-accelerated Databases. In *SIGMOD*, pages 1891–1906, 2016.
[4] P. Carbone, A. Katsifodimos, et al. Apache Flink$^{\text{TM}}$: Stream and Batch Processing in a Single Engine. *IEEE Data Eng. Bull.*, pages 28–38, 2015.
[5] B. Gedik. Partitioning Functions for Stateful Data Parallelism in Stream Processing. *VLDB*, pages 517–539, 2014.
[6] H. Hacigumus, Y. Chi, et al. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable? In *ICDE*, pages 1081–1092, 2013.
[7] J. He, M. Lu, and B. He. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *VLDB*, pages 889–900, 2013.
[8] M. Hosomi, H. Yamagishi, et al. A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-RAM. In *IEDM*, pages 459–462, 2005.
[9] J. Kraemer. *Continuous Queries over Data Streams - Semantics and Implementation.* PhD thesis, Philipps-Universitaet Marburg, 2007.
[10] B. C. Lee, E. Ipek, et al. Architecting Phase Change Memory As a Scalable Dram Alternative. ISCA, pages 2–13, 2009.
[11] V. Leis, A. Gubichev, et al. How Good Are Query Optimizers, Really? *VLDB*, pages 204–215, 2015.
[12] Y. Liu, M. Mortazavi, et al. Cost-Based Data-Partitioning for Intra-Query Parallelism. In *DB&IS*, 2014.
[13] S. Manegold. *Understanding, Modeling, and Improving Main-Memory Database Performance.* PhD thesis, Universiteit van Amsterdam, 2002.
[14] C. Sirish, C. Owen, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 46–58, 2003.
[15] D. B. Strukov, G. S. Snider, et al. The missing memristor found. *Nature*, pages 80–83, 2008.
[16] A. Toshniwal, S. Taneja, et al. Storm@Twitter. In *SIGMOD*, pages 147–156, 2014.
[17] S. D. Viglas and J. F. Naughton. Rate-based Query Optimization for Streaming Information Sources. SIGMOD, pages 37–48, 2002.