Analyzing Efficient Stream Processing on Modern Hardware

Steffen Zeuch [2], Bonaventura Del Monte [2], Jeyhun Karimov [2], Clemens Lutz [2],
Manuel Renz [2], Jonas Traub [1], Sebastian Breß [1,2], Tilmann Rabl [1,2], Volker Markl [1,2]
[1]Technische Universität Berlin, [2]German Research Center for Artificial Intelligence
firstname.lastname@dfki.de

ABSTRACT

Modern *Stream Processing Engines* (SPEs) process large data volumes under tight latency constraints. Many SPEs execute processing pipelines using message passing on shared-nothing architectures and apply a partition-based *scale-out* strategy to handle high-velocity input streams. Furthermore, many state-of-the-art SPEs rely on a Java Virtual Machine to achieve platform independence and speed up system development by abstracting from the underlying hardware.

In this paper, we show that taking the underlying hardware into account is essential to exploit modern hardware efficiently. To this end, we conduct an extensive experimental analysis of current SPEs and SPE design alternatives optimized for modern hardware. Our analysis highlights potential bottlenecks and reveals that state-of-the-art SPEs are not capable of fully exploiting current and emerging hardware trends, such as multi-core processors and high-speed networks. Based on our analysis, we describe a set of design changes to the common architecture of SPEs to *scale-up* on modern hardware. We show that the single-node throughput can be increased by up to two orders of magnitude compared to state-of-the-art SPEs by applying specialized code generation, fusing operators, batch-style parallelization strategies, and optimized windowing. This speedup allows for deploying typical streaming applications on a single or a few nodes instead of large clusters.

PVLDB Reference Format:
Steffen Zeuch, Bonaventura Del Monte, Jeyhun Karimov, Clemens Lutz, Manuel Renz, Jonas Traub, Sebastian Breß, Tilmann Rabl, Volker Markl. Analyzing Efficient Stream Processing on Modern Hardware. *PVLDB*, 12(5): xxxx-yyyy, 2019.
DOI: https://doi.org/10.14778/3303753.3303758.

1. INTRODUCTION

Over the last decade, streaming applications have emerged as an important new class of big data processing use cases. Streaming systems have to process high velocity data streams under tight latency constraints. To handle high velocity streams, modern SPEs such as Apache Flink [29], Apache Spark [91], and Apache Storm [81] distribute processing over a large number of computing nodes in a cluster, i.e., *scale-out* the processing. These systems trade single node performance for scalability to large clusters and use a Java Virtual Machine (JVM) as the underlying processing environment for platform independence. While JVMs provide a high level of abstraction from the underlying hardware, they cannot easily provide efficient data access due to processing overheads induced by data (de-)serialization, objects scattering in main memory, virtual functions, and garbage collection. As a result, the overall performance of scale-out SPEs building on top of a JVM are severely limited in throughput and latency. Another class of SPEs optimize execution to *scale-up* the processing on a single node. For example, Saber [58], which is build with Java, Streambox [67], which is build with C++, and Trill [34], which is build with C#.

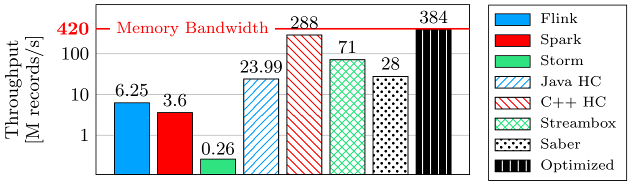


Figure 1: Yahoo! Streaming Benchmark (1 Node).

In Figure 1, we show the throughput of modern SPEs executing the Yahoo! Streaming Benchmark [39] on a single node compared to hand-coded implementations. We report the detailed experimental setup in Section 5.1. As shown, the state-of-the-art SPEs Apache Flink, Spark, and Storm achieve sub-optimal performance compared to the physical limit that is established by the memory bandwidth. To identify bottlenecks of SPEs, we hand-code the benchmark in Java and C++ (HC bars). Our hand-coded Java implementation is faster than Flink but still 40 times slower than the physical limit. In contrast, our C++ implementation and Streambox omit serialization, virtual functions, and garbage collection overheads and store tuples in dense arrays (arrays of structures). This translates to a 26 times better performance for our C++ implementation and a 6 times better performance for Streambox compared to the Java implementation. Finally, we show an *optimized* implementation that eliminates the potential SPE-bottlenecks which we identify in this paper. Our optimized implementation operates near the physical memory limit and utilizes modern hardware efficiently.

The observed sub-optimal single node performance of current SPEs requires a large cluster to achieve the same performance as a single node system using the scale-up optimizations that we evaluate in this paper. The major advantage of a scale-up system is the avoidance of inter-node

Proceedings of the VLDB Endowment, Vol. 12, No. 5
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3303753.3303758.

data transfer and a reduced synchronization overhead. With hundreds of cores and terabytes of main memory available, modern scale-up servers provide an interesting alternative to process high-volume data streams with high throughput and low latency. Utilizing the capabilities of modern hardware on a single node efficiently, we show that the throughput of a single node becomes sufficient for many streaming applications and can outperform a cluster of several nodes. As one result of this paper, we show that SPEs need to be optimized for modern hardware to exploit the possibilities of emerging hardware trends, such as multi-core processors and high-speed networks.

We investigate streaming optimizations by examining different aspects of stream processing regarding their exploitation of modern hardware. To this end, we adopt the terminology introduced by Hirzel et al. [53] and structure our analysis in data-related and processing-related optimizations. As a result, we show that by applying appropriate optimizations, single node throughput can be increased by two orders of magnitude. Our contributions are as follows:

- We analyze current SPEs and identify their inefficiencies and bottlenecks on modern hardware setups.
- We explore and evaluate new architectures of SPEs on modern hardware that address these inefficiencies.
- Based on our analysis, we describe a set of design changes to the common architecture of SPEs to *scale-up* on modern hardware.
- We conduct an extensive experimental evaluation using the Yahoo! Streaming Benchmark, the Linear Road Benchmark, and a query on the NY taxi data set.

The rest of this paper is structured as follows. In Section 2, we introduce concepts of current SPEs as well as aspects of modern hardware that we investigate in this paper. After that, we explore the data-related (Section 3) and processing-related stream processing optimization (Section 4) of an SPE on modern hardware. In Section 5, we evaluate state-of-the-art SPEs and investigate different optimizations in an experimental analysis. Finally, we discuss our results in Section 5.3 and present related work in Section 6.

2. BACKGROUND

Stream processing engines enable the processing of long-running queries over unbounded, continuous data streams. A major challenge for SPEs is to provide near real-time processing guarantees. Furthermore, they have to handle fluctuations in the input data rate. In contrast, *batch processing engines* pull data from storage as needed, which allows for controlling input rates [14, 29]. In this section, we introduce the underlying processing models of SPEs as well as the SPEs that we selected for our study (Section 2.1). After that, we present aspects of modern hardware that we investigate in this paper (Section 2.2).

2.1 Streaming Systems

Over the last decades, two categories of streaming systems emerged. SPEs in the first category are optimized for *scale-out* execution of streaming queries on shared-nothing architectures. In general, these SPEs apply a distributed producer-consumer pattern and a buffer mechanism to handle data shuffling among operators (e.g., partitioning). Their goal is to massively parallelize the workload among many small to medium sized nodes. In this paper, we analyze Apache Flink [29], Apache Spark [91], and Apache Storm [81] as representative, JVM-based, state-of-the-art scale-out SPEs. We choose these SPEs due to their maturity, wide academic and industrial adoption, and the size of their open-source communities.

SPEs in the second category are optimized for *scale-up* execution on a single machine. Their goal is to exploit the capabilities of one high-end machine efficiently. Recent SPEs in this category are Streambox [67], Trill [34], and Saber [58]. In particular, Streambox aims to optimize the execution for multi-core machines, whereas SABER takes heterogeneous processing into account by utilizing GPUs. Finally, Trill supports a broad range of queries beyond SQL-like streaming queries. In this paper, we examine Streambox and SABER as representative SPEs in this category.

A major aspect of parallel stream processing engines are the underlying processing models which are either based on micro-batching or use pipelined tuple-at-a-time processing [14, 19, 33, 42, 89]. *Micro-batching* SPEs split streams into finite chunks of data (batches) and process these batches in parallel. SPEs such as Spark [90], Trident [1], and SABER [58] adopt this micro-batching approach. In contrast, *pipelined, tuple-at-a-time* systems execute data-parallel pipelines consisting of stream transformations. Instead of splitting streams into micro-batches, operators receive individual tuples and produce output tuples continuously. SPEs such as Apache Flink [15] and Storm [81] adopt this approach.

2.2 Modern Hardware

A streaming system on modern hardware has to exploit three critical hardware resources efficiently to achieve a high resource utilization: CPU, main memory, and network. In this paper, we investigate if current SPEs and their system designs exploit these resources efficiently.

First, modern *many-core CPUs* contain dozens of CPU cores per socket. Additionally, multi-socket CPUs connect multiple CPUs via fast interconnects to form a network of cores. However, this architecture introduces non-uniform memory access (NUMA) among cores [62, 74]. As a result, the placement of data in memory directly affects the overall efficiency of the system. Furthermore, each core caches parts of the data in its private and shared caches for fast access. To exploit this performance critical resource efficiently, the data and instruction locality have to be taken into account.

Second, today's common servers provide *main memory* capacities of several terabytes. As a result, the majority of the data and state informations, e.g., the data of one window, can be maintained in main memory. In combination with multiple threads working on the same data, this significantly speeds up the processing compared to maintaining the same state with slower disk or network connections. On the other hand, such a streaming system has to handle synchronization and parallel aspects in much shorter time frames to enable efficient processing.

Third, today's network connections become constantly faster and potentially outperform main memory bandwidth [22] in the future. Commonly available Ethernet technologies provide 1, 10, 40, or 100 Gbit bandwidth. In contrast, new network technologies such as InfiniBand provide much higher bandwidth up to or even faster than main memory bandwidth [22]. This important trend will lead to radical changes in system designs. In particular, the common wisdom of processing data *locally first* does not hold with future network technologies [22]. As a result, a system which transfers data between nodes as fast as or even faster than reading from main memory introduces new challenges and opportunities to future SPEs.

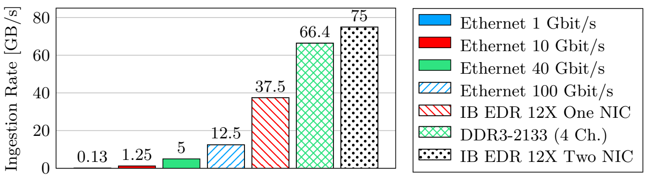


Figure 2: Data ingestion rate overview.

3. DATA-RELATED OPTIMIZATIONS

In this section, we explore data-related aspects for streaming systems. We discuss different methods to supply high velocity streams in Section 3.1 and evaluate strategies for efficient data passing among operators in Section 3.2.

3.1 Data Ingestion

We examine different alternatives how SPEs can receive input streams. In Figure 2, we provide an overview of common ingestion sources and their respective maximum bandwidths. In general, an SPE receives data from network, e.g., via sockets, distributed file systems, or messaging systems. Common network bandwidths range from 1 to 100 GBit over Ethernet (125 MB/s - 12.5 GB/s). In contrast, InfiniBand (IB) offers higher bandwidths from 1.7 GB/s (FDR 1x) to 37.5 GB/s (EDR - Enhanced Data Rate - 12x) per network interface controller (NIC) per port [3].

Analysis. Based on the numbers in Figure 2, we conclude that modern network technologies enable ingestion rates near main memory bandwidth or even higher [22]. This is in line with Binning et al. who predict the end of slow networks [22]. Furthermore, they point out that future InfiniBand standards such as HDR or NDR will offer even higher bandwidths. Trivedi et al. [84] assess the performance-wise importance of the network for modern distributed data processing systems such as Spark and Flink. They showed that increasing the network bandwidth from 10 Gbit to 40 Gbit transforms those systems from network-bound to CPU-bound systems. As a result, improving the single node efficiency is crucial for scale-out systems as well. On single node SPEs, Zhang et al. [94] point out that current SPEs are significantly CPU-bound and thus will not natively benefit from an increased ingestion rate. An SPE on modern hardware should be able to exploit the boosted ingestion rate to increase its overall, real-world throughput.

Infiniband and RDMA. The trends in the area of networking technologies over the last decade showed, that Remote Direct Memory Access (RDMA) capable networks became affordable and thus are present in an increasing number of data centers [56]. RDMA enables the direct access to main memory of a remote machine while bypassing the remote CPU. Thus, it does not consume CPU resources of the remote machine. Furthermore, the caches of the remote CPU are not polluted with the transfered memory content. Finally, *zero-copy* enables direct send and receive using buffers and bypasses the software network stack. Alternatively, high-speed networks based on Infiniband can run common network protocols such as TCP and UDP (e.g., via IPoIB). However, this removes the benefits of RDMA because their socket interfaces involve system calls and data copies between application buffers and socket buffers [75].

Today, three different network technologies provide RDMA-support: 1) InfiniBand, 2) RoCE (RDMA over Converged Ethernet), and 3) iWARP (Internet Wide Area RDMA Protocol). The underlying NICs are able to provide up to 100 Gbps of per-port bandwidth and a round-trip latency of roughly 2μs [56]. Overall, there are two modes for RDMA communication: one-sided and two-sided. With two-sided communication, both sides, the sender and the receiver are involved in the communication. Thus, each message has to be acknowledged by the receiver and both participants are aware that a communication takes place. In contrast, one-sided communication involves one active sender and one passive receiver (RDMA write) or a passive sender and an active receiver (RDMA read). In both cases, one side is agnostic of the communication. As a result, synchronization methods are required to detect that a communication is finished.

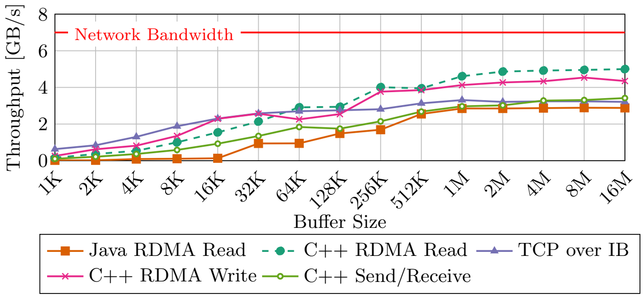


Figure 3: Infiniband network bandwidth.

Experiment. In Figure 3, we run the aforementioned communication modes on an InfiniBand high-speed network with a maximum bandwidth of 56 Gbit/s (FDR 4x). On the x-axis, we scale the buffer/message and on the y-axis we show the throughput in GBytes/s for a data transfer of 10M tuples of 78 byte each, i.e., the yahoo streaming benchmark tuples. To implement the C++ version, we use RDMA Verbs [40]. The Java implementation uses the *disni* library [2] and follows the Spark RDMA implementation [6]. As a first observation, Figure 3 shows that all modes perform better with larger buffer/messages sizes and their throughput levels off starting around 1M. Second, the C++ RDMA read implementation achieves roughly 2 GB/s more throughput than the Java implementation. As a result, we conclude that the JVM introduces additional overhead and, thus, a JVM-based SPE is not able to utilize the entire network bandwidth of modern high-speed networks. Third, the C++ RDMA write operation is slightly slower than the read operation. Finally, the two-sided send/receive mode is slower than the one-sided communication modes. Overall, the one-sided RDMA read communication using a message size of 4 MB performs best. Note that, our findings are in line with previous work [68, 56].

Discussion. Our experiments show that a JVM-based distributed implementation is restricted by the JVM and cannot utilize the bandwidth as efficiently as a C++ implementation. As a result, the performance of JVM-based SPEs is limited in regards to upcoming network technologies that are able to deliver data at memory-speed.

3.2 Data Exchange between Operators

In this section, we examine the design of pipelined execution using message passing, which is common in state-of-the-art SPEs. In particular, SPEs execute data-parallel pipelines using asynchronous message passing to provide high-throughput processing with low-latency. To pass data among operators, they use queues as a core component. In general, data is partitioned by keys and distributed among parallel instances of operators. In this case, queues enable asynchronous processing by decoupling producer and consumer. However, queues become the bottleneck if operators overload them. In particular, a slow consumer potentially causes back-pressure and slows down previous operators.

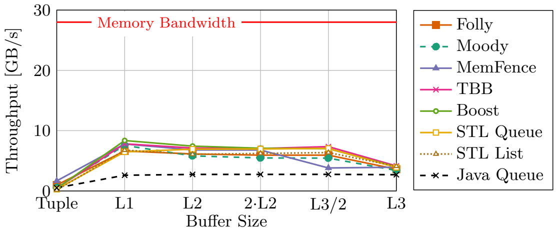


Figure 4: Queue throughput.

To examine queues as the central building block in modern SPEs, we conduct an experiment to asses the capabilities of open-source, state-of-the-art queue implementations.

Experimental Setup. In this experiment, we selected queues that cover different design aspects such as in memory management, synchronization, and underlying data structures. We select queues provided by Facebook (called Folly) [7], the Boost library [8], the Intel TBB library [9], and the Java *java.util.concurrent* package [4]. In addition, we choose an implementation called *Moody* [5] and an implementation based on memory fences [85]. Finally, we provide our own implementations based on the STL queue and list templates. Except the Java queue, all queues are implemented in C++.

Following Sax et al. [55] and Carbone et al. [29], we group tuples into buffers to increase the overall throughput. They point out, that buffers introduce a trade-off between latency and throughput. For our C++ implementations, we enqueue filled buffers into the queue by an explicit copy and dequeue the buffer with a different thread. In contrast, our Java queue implementation follows Flink by using buffers as an unmanaged memory area. A tuple matches the size of a cache line, i.e., 64 Byte. We pass 16.7M tuples, i.e., 1 gigabyte of data, via the queue.

Observation. In Figure 4, we relate the throughput measurements in GB/s to the physically possible main memory bandwidth of 28 GB/s. Furthermore, we scale the *buffer size* on the x-axis such that it fits a particular cache size. In general, the lock-free queues from the TBB and Boost library achieve the highest throughputs of up to 7 GB/s, which corresponds to 25% of the memory bandwidth. Other C++ implementations achieve slightly lower throughput of up to 7 GB/s. In contrast, the Java queue, which uses conditional variables, achieves the lowest throughput of up to 3 GB/s. Overall, buffering improves throughput as long as the buffer size either matches a private cache size (L1 or L2) or is smaller than L3 cache size. Inside this range, all buffer sizes perform similar with a small advantage for a buffer size that matches the L1 cache.

Discussion. Figure 4 shows that queues are a potential bottleneck for a pipelined streaming system using message passing. Compared to a common memory bandwidth of approximately 28 GB/s, the best state-of-the-art queue implementations exploit only one fourth of this bandwidth. Furthermore, a Java queue achieves only 10% of the available memory bandwidth because it exploits conditional variables and unmanaged buffers. These design aspects introduce synchronization as well as serialization overhead.

As a result, the maximum throughput of a streaming system using queues is bounded by the number of queues and their aggregated maximum throughput. Additionally, load imbalances reduce the maximum throughput of the entire streaming system. In particular, a highly skewed data stream can overload some queues and under-utilize others.

3.3 Discussion

The results in Figure 3, suggest to use RDMA read operations and a buffer size of several megabytes to utilize todays high-speed networks in a distributed SPE. The results in Figure 4 strongly suggest that a high performance SPE on modern hardware should omit queues as a potential bottleneck. If queues are used, the buffer size should be between the size of the L1 and 2*L2 cache.

4. PROCESSING-RELATED OPTIMIZATIONS

In this section, we examine the processing-related design aspects of SPEs on modern hardware. Following the classification of stream processing optimizations introduced by Hirzel et al. [53], we address operator fusion in Section 4.1, operator fission in Section 4.2, and windowing techniques in Section 4.3. We will address the stream-processing optimization of *placement* and *load balancing* in the context of operator fusion and fission [53].

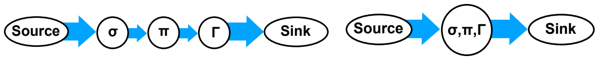
4.1 Operator Fusion

We examine two different execution models for SPEs, namely *query interpretation* and *compilation-based query execution*. Both models enable data-parallel pipelined stream processing. For our following considerations, we assume a query plan that is translated into a processing pipeline consisting of sources, intermediate operators, and sinks. Sources continuously provide input data to the processing pipeline and sinks write out the result stream. In general, we can assign a different degree of parallelism to each operator as well as to each source and sink.

4.1.1 Interpretation-based Query Execution

An interpretation-based processing model is characterized by: 1) an *interpretation-based evaluation* of the query plan, 2) the *application of queues* for data exchange, and 3) the possibility of *operator fusion*. In Figure 5(a), we show an example query plan of a simple select-project-aggregate query using a query interpretation model. To this end, a query plan is translated into a set of pipelines containing producer-consumer operators. Each operator has its own processing function which is executed for each tuple. The interpretation-based processing model uses queues to forward intermediate results to downstream operators in the pipeline. Each operator can have multiple instances where each instance reads from an input queue, processes a tuple, and pushes its result tuples to its output queue. Although all operators process a tuple at a time on a logical level, the underlying engine might perform data exchanges through buffers (as shown in Section 3.2).

In general, interpretation-based processing is used by modern SPEs like Flink, Spark, and Storm to link parallel operator instances using a push-based approach implemented by function calls and queues. In case an operator instance forwards its results to only one subsequent operator, both operators can be fused. Operator fusion combines operators into a single tight *forloop* that passes tuples via register and thus eliminating function calls. Note, that this register-level operator fusion goes beyond function call-based operator fusion used in other SPS [30, 80, 48]. However, many operators like the aggregate-by-key operator in Figure 5(a) require a data exchange due to partitioning. In particular, each parallel instance of the aggregate-by-key operator processes a



(a) Interpretation-based. (b) Compilation-based.

Figure 5: Query execution strategies.

range of keys and stores intermediate results as internal operator state. As a result, queues are a central component in an interpretation-based model to link operators. Note that, this execution model represents each part of the pipeline as a logical operator. To execute a query, every logical operator is mapped to a number of physical operators, which is specified by the assigned degree of parallelism.

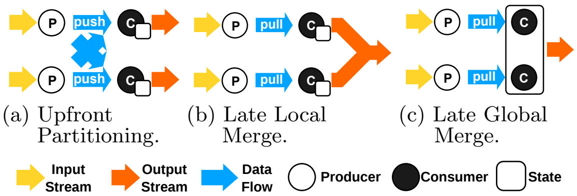
4.1.2 Compilation-based Query Execution

A compilation-based processing model follows the ideas of Neumann [70] and is characterized by: 1) the execution of compiled query plans (using custom code generation), 2) operator fusion performed at query compile time, and 3) an improved hardware resource utilization. In general, this approach fuses operators within a pipeline until a so-called *pipeline-breaker* is reached. A pipeline breaker requires the full materialization of its result before the next operator can start processing. In the context of databases and in-memory batch processing, pipeline breakers are relational operators such as sorting, join, or aggregations. In the context of streaming systems, the notion of a pipeline breaker is different because streaming data is unbounded and, thus, the materialization of a full stream is impracticable. Therefore, we define operators that send partially materialized data to the downstream operator as *soft* pipeline breakers. With *partially materialized*, we refer to operators that require buffering before they can emit their results, e.g., windowed aggregation. Compilation-based SPEs such as IBM's System S/SPADE and SABER use operator fusion to remove unnecessary data movement via queues.

In Figure 5(b), we present a select-project-aggregate query in a compilation-based model. In contrast to an interpretation-based model, all three operators are fused in a single operator that can run with any degree of parallelism. This model omits queues for message passing and passes tuples via CPU registers. Additionally, it creates a tight loop over the input stream, which increases data and instruction locality. Neumann showed that this model leads to improved resource utilizations on modern CPUs in databases [70]. In terms of parallelization, operator fusion results in a reduced number of parallel operators compared to the query interpretation-based model. Instead of assigning one thread to the producer and another thread to the consumer, this model executes both sides in one thread. Thus, with the same number of available threads, the parallelism increases.

4.1.3 Discussion

We contrast two possible processing models for an SPE. On a conceptual level, we conclude that the compilation-based execution model is superior for exploiting the resource capabilities of modern hardware. In particular, it omits queues, avoids function calls, and results in code optimized for the underlying hardware. We expect the same improvement for streaming queries as Neumann reported for database queries [70]. However, a scale-out optimized SPE commonly introduces queues to distribute the computation among different nodes to decouple the producer from the consumer.



(a) Upfront Partitioning. (b) Late Local Merge. (c) Late Global Merge.

Figure 6: Parallelization strategies.

4.2 Operator Fission

In this section, we present two general parallelization strategies to distribute the processing among different computing units. This stream processing optimization includes partitioning, data parallelism, and replication [53]. With *Upfront Partitioning* we refer to a strategy that physically partitions tuples such that each consumer processes a distinct sub-set of the stream. With *Late Merging* we refer to a strategy that logically distributes chunks of the stream to consumers, e.g., using a round-robin. This strategy may require a final merge step at the end of the processing if operators such as keyed aggregations or joins are used.

Upfront Partitioning (UP). Upfront Partitioning introduces a physical partitioning step between producer and consumer operators. This partitioning consists of an intermediate data shuffling among n producers and m consumers, which are mapped to threads. Each producer applies a partitioning function (e.g., consistent hashing or round-robin) after processing a tuple to determine the responsible consumer. In particular, partitioning functions ensure that tuples with the same key are distributed to the same consumer. In Figure 6(a), we present this common partition-based parallelization strategy, which is used by Apache Flink [29], Apache Spark Streaming [91], and Apache Storm [81].

This strategy leverages queues to exchange data among producer and consumer operators. Thus, each producer explicitly pushes its output tuples to the input queue of its responsible consumer. Although the use of buffers improves the throughput of this strategy (see Section 3.2), we showed that queues establish a severe bottleneck in a scale-up SPE. Furthermore, the load balancing on consumer side highly depends on the quality of the partitioning function. If data are evenly distributed, the entire system achieves high throughput. However, an uneven distribution leads to over-/underutilized parallel instances of an operator.

The processing using upfront partitioning creates independent/distinct, thread-local intermediate results on each consumer. Thus, assembling the final output result requires only a concatenation of thread local intermediate results. An SPE can perform this operation in parallel, e.g., by writing to a distributed file system.

Late Merging. Late Merging divides the processing among independent threads that pull data by their own. Leis et al. [62] and Pandis et al. [72] previously applied this partitioning approach in the context of high-performance, main memory databases. In this paper, we revise those techniques in the context of stream processing. In contrast to upfront partitioning, this strategy requires an additional merge step at the end of the processing pipeline. In the following, we introduce two merging strategies: *Late Local Merge* and *Late Global Merge*.

In the **Late Local Merge (LM)** strategy, each worker thread buffers its partial results in a *local* data structure (see Figure 6(b)). When the processing pipeline reaches a soft pipeline-breaker (e.g., a windowed aggregation), those partial results have to be merged into a global data structure. Because each worker potentially has tuples of the entire domain, the merge step is more complex compared to the simple concatenation used by UP. One solution for merging the results of a soft pipeline-breaker such as an aggregation operator is to use parallel tree aggregation algorithms.

In the **Late Global Merge (GM)** strategy, all worker threads collaboratively create a global result during execution (see Figure 6(c)). In contrast to late local merge, this strategy omits an additional merging step at the end. Instead, this strategy introduces additional synchronization overhead during run-time. In particular, if the query plan contains stateful operators, e.g., keyed aggregation, the system has to maintain a global state among all stateful operators of a pipeline. To minimize the maintenance costs, a lock-free data structure should be utilized. Furthermore, contention can be mitigated among instances of stateful operators through fine-grained updates using atomic compare-and-swap instructions. Finally, Leis et al. [62] describe a similar technique in the context of batch processing, whereas Fernandez et al. [45] provide solutions for scale-out SPEs.

Depending on the characteristics of the streaming query, an SPE should either select late merge or global merge. If the query induces high contention on a small set of data values, e.g., an aggregation with a small group cardinality, local merge is beneficial because it reduces synchronization on the global data structure. In contrast, if a query induces low contention, e.g., an aggregation with a large group cardinality, global merge is beneficial because it omits the additional merging step at the end.

Discussion. In this section, we presented different parallelization strategies. UP builds on top of queues to exchange data between operators. This strategy is commonly used in scale-out SPEs in combination with an interpretation-based model. However, as shown in Section 3.2, queues are a potential bottleneck for a scale-up optimized SPE. Therefore, we present two late merging strategies that omit queues for data exchange. In particular for a scale-up SPE, those strategies enable a compilation-based execution (see Section 4.1). With late merging and a compilation-based query execution, we expect the same improvements for SPEs that Neumann achieved for in-memory databases [70].

4.3 Windowing

Windowing is a core feature of streaming systems that splits the conceptually infinite data stream into finite chunks of data, i.e., *windows*. An SPE computes an aggregate for each window, e.g., the revenue per month. As a main goal, an efficient windowing mechanism for an SPE on modern hardware has to minimize the synchronization overhead to enable a massively parallel processing. To this end, we implement a lock-free continuous windowing mechanism following the ideas in [58, 78, 79]. Our implementation uses a double-buffer approach and fine-grained synchronization primitives, i.e., atomic instructions, instead of coarse-grained locks to minimize synchronization overhead. Moreover, our implementation induces a smaller memory footprint than approaches based on aggregate trees [17] as we use on-the-fly aggregation [79] instead of storing aggregate trees.

Alternating Window Buffers. In Figure 7, we show our double-buffer implementation based on the ideas in [58, 78, 79]. At its core, it exploits alternating window buffers to ensure that there is always one *active buffer* as a destination of incoming tuples. Furthermore, multiple non-active buffers store the results of previous windows. The non-active buffers can be used to complete the aggregate computation, output the results, and reinitialize the buffer memory. As a result, this implementation never defers the processing of the input stream, which increases the throughput of the window operation for both parallelization strategies (see Section 4.2). To implement the buffers, we use lock-free data structures that exploit atomic compare-and-swap instructions. This enables concurrent accesses and modification [51, 62, 63]. As a result, several threads can write to a window buffer in parallel and thereby aggregate different parts.

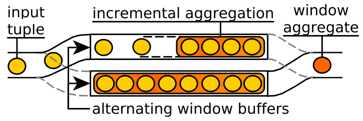


Figure 7: Alternating window buffers.

Detecting Window Ends. We implement an event-based approach where writer threads check upon the arrival of a tuple if the current window ended. This implies a very low latency for high-bandwidth data streams because arriving tuples cause frequent checks for window ends. Discontinuous streams [60] may lead to high latencies, because of the absence of those checks during the discontinuity. In this case, we switch between our lock-free implementation and a timer-thread implementation depending on the continuity of input streams. A timer thread checks for window ends periodically and, if required, triggers the output. We implement the switch between timer threads and our event-driven technique directly at the source by monitoring the input rate.

Extensions. In order to achieve very low output latencies, we aggregate tuples incrementally whenever possible [79]. To further improve performance, we can combine our implementation with stream slicing [31, 61, 64, 82, 83]. Slicing techniques divide a data stream in non-overlapping chunks of data (*slices*) such that all windows are combinations of slices. When processing sliding windows, we switch alternating buffers upon the start of each new slice. We use time-based windows as an example to simplify the description. However, our windowing technique works with arbitrary stream slicing techniques, which enables diverse window types such as count-based windows [12, 52] and variants of data-driven and user-defined windows [31, 47, 50].

Discussion. We implement a lock-free windowing mechanism that minimizes the contention between worker threads. This minimized contention is crucial for a scale-up optimized SPE on modern hardware. To determine its efficiency, we run a micro-benchmark using the data set of the Yahoo! Streaming Benchmark and compare it to a common solution based on timer threads. Our results show that a double-buffer approach achieves about 10% higher throughput on a scale-up SPE on modern hardware.

5. EVALUATION

In this section, we experimentally evaluate design aspects of an SPE on modern hardware. In Section 5.1, we introduce our experimental setup including machine configuration and selected benchmarks. After that, we conduct a series of experiments in Section 5.2 to understand the reasons for different throughput values of different implementations and state-of-the-art SPEs. Finally, we summarize and discuss our results in Section 5.3.

5.1 Experimental Setup

In the following, we present the hardware and software configurations used for our analysis as well as the selected benchmarks and their implementation details.

Hardware and Software. We execute our benchmarks on an Intel Core i7-6700K processor with 4 GHz and four physical cores (eight cores using hyper-threading). This processor contains a dedicated 32 KB L1 cache for data and instructions per core. Additionally, each core has a 256 KB L2 cache and all cores share an 8 MB L3 cache. The test system has 32 GB of main memory and runs Ubuntu 16.04. If not stated otherwise, we execute all measurements using all logical cores, i.e., a degree of parallelism of eight.

The C++ implementations are compiled with GCC 5.4 and O3 optimization as well as the *mtune* flags to produce specific code for the underlying CPU. The Java implementations run on the HotSpot VM in version 1.8.0_131. We use Apache Flink 1.3.2, Apache Spark Streaming 2.2.0, and Apache Storm 1.0.0 as scale-out SPEs. We disable fault-tolerance mechanisms (e.g., checkpointing) to minimize the overhead of those frameworks. We use the Streambox (written in C++) release of March 10th and Saber (written in Java) in version 0.0.1 as representative scale-up SPEs. We measure hardware performance counters using Intel VTune Amplifier XE 2017 and the PAPI library 5.5.1.

The JVM-based scale-out SPEs have to serialize and deserialize tuples to send them over the network. The (de)-serialization requires a function call and a memory copy from/to the buffer for each field of the tuple, which adds extra overhead for JVM-based SPEs [84]. In contrast, a scale-up C++ based SPE accesses tuples directly in dense in-memory data structures.

Strategies Considered. We implement each benchmark query using the three previously discussed parallelization strategies: upfront partitioning as well as late merging with Local Merge and Global Merge (see Section 4.2). The upfront partitioning uses an interpretation-based execution strategy and both Late Merge implementations use a compilation-based execution. Furthermore, all implementations use the lock-free windowing mechanism described in Section 4.3. For each strategy and benchmark, we provide a Java and a C++ implementation.

The upfront partitioning strategy follows the design of state-of-the-art scale-out SPEs. Thus, it uses message passing via queues to exchange tuples among operators. In contrast, both Late Merge strategies omit queues and fuse operators where possible. Furthermore, we pass tuples via local variables in CPU registers. In the following, we refer to Late Local Merge and Late Global Merge as Local Merge (LM) and Global Merge (GM), respectively.

In our experiments, we use the memory bandwidth as an upper bound for the input rate. However, future network technologies will increase this boundary [22]. To achieve a data ingestion at memory speed, we ingest streams to our implementations using an in-memory structure. Each in-memory structure is thread-local and contains a pre-generated, disjoint partition of the input. We parallelize query processing by assigning each thread to a different input partition. At run-time, each implementation reads data from an in-memory structure, applies its processing, and outputs its results in an in-memory structure through a sink. Thus, we exclude network I/O and disks from our experiments.

Java Optimizations. We optimize our Java implementations and the JVM to overcome well-known performance issues. We ensure that the just-in-time compiler is warmed-up [66] by repeating the benchmark 30 times. We achieve efficient memory utilization by avoiding unnecessary memory copy, object allocations, GC cycles, and pointer chasing operations [28]. Furthermore, we set the heap size to 28 GB to avoid GC overhead, use primitive data types and byte buffer for pure-java implementations and off-heap memory for the scale-out SPEs [43, 87]. Finally, we use the Garbage-First (G1GC) garbage collector which performs best for our examined benchmarks [10].

Benchmarks. We select the Yahoo! Streaming Benchmark (YSB), the Linear Road Benchmark (LRB), and a query on the New York City Taxi (NYT) dataset to assess the performance of different design aspects for modern SPEs. YSB simulates a real-word advertisement analytics task and its main goal is to assess the performance of windowed aggregation operators [38, 39]. We implement YSB [38] using 10K campaigns based on the codebase provided by [49, 39]. Following these implementations, we omit a dedicated key/value store and perform the join directly in the engine [49, 39].

LRB simulates a highway toll system [16]. It is widely adopted to benchmark diverse systems such as relational databases [16, 25, 65, 77], specialized streaming systems [12, 16, 54, 46], distributed systems [24, 92, 93], and cloud-based systems [32, 37]. We implement the accident detection and the toll calculation for the LRB benchmark [16] as a sub-set of the queries.

The New York City Taxi dataset covers 1.1 billion individual taxi trips in New York from January 2009 through June 2015 [76]. Besides pickup and drop-off locations, it provides additional informations such as the trip time or distance, the fare, or the number of passengers. We divided the area of NY into 1000 distinct regions and formulate the following business relevant query: *What are the number of trips and their average distance for the VTS vendor per region for rides more than 5 miles over the last two seconds?* The answer to this query would provide a value information for taxi drivers in which region they should cruise to get a profitable ride.

YSB, LRB, and NYT are three representative workloads for todays SPEs. However, we expect the analysis of other benchmarks would obtain similar results. We implement all three benchmarks in Flink, Storm, Spark Streaming, Streambox, and Saber. Note that for SABER, we use the YSB implementation provided in [73] and implement the LRB benchmark and NYT query accordingly. Finally, we provide hand-coded Java and C++ implementations for the three parallelization strategies presented in Section 4.2. In our experiments, we exclude the preliminary step of preloading data into memory from our measurements.

Experiments. In total, we conduct six experiments. First, we evaluate the throughput of the design alternatives showed in this paper to get an understanding of their overall performance. In particular, we examine the throughput of the design alternatives for the YSB and LRB (see Section 5.2.1). Second, we break down the execution time of the design alternatives to identify the CPU component that consumes the majority of the execution time (see Section 5.2.2). Third, we sample data and cache-related performance counters to compare the resource utilization in detail (see Section 5.2.3). Fourth, we compare our scale-out optimized implementations of the YSB with published benchmark results of state-of-the-art SPEs running in a cluster

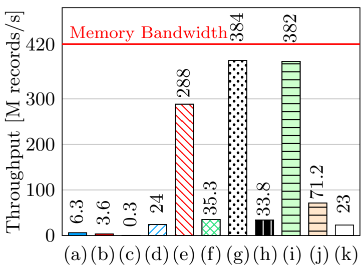


Figure 8: YSB single node.

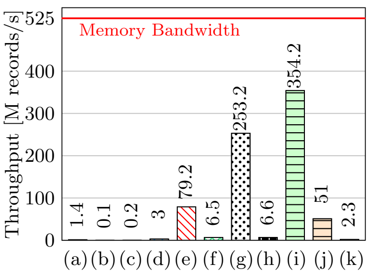


Figure 9: LRB single node.

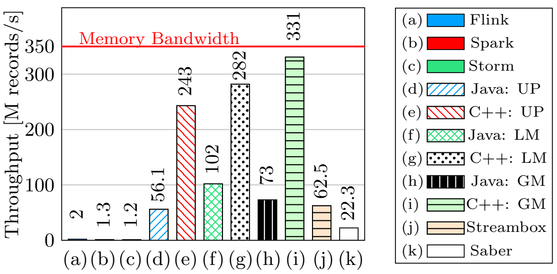


Figure 10: NYT query single node.

(see Section 5.2.4). Fifth, we show how scale-out optimizations perform in a distributed execution using a high-speed InfiniBand in Section 5.2.5. Finally, we evaluate the impact of design alternatives on latency (see Section 5.2.6).

5.2 Results

In this section, we present the results of the series of experiments presented in the previous section. The goal is to understand the differences in performance of state-of-the-art SPEs and design alternatives presented in this paper.

5.2.1 Throughput

To understand the overall efficiency of the design alternatives presented in this paper, we measure their throughput in million records per second for both benchmarks in Figures 8 and 9. Additionally, we relate their throughput to the physically possible memory bandwidth, which could be achieved by only reading data from an in-memory structure without processing it.

YSB Benchmark. The scale-out optimized SPEs Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 1.5% (Flink), 0.8% (Spark Streaming), and 0.07% (Storm). The scale-up optimized Streambox (written in C++) achieves about 17% and Saber (written in Java) achieves 6.6% of the theoretically possible throughput. In contrast, the C++ implementations achieve a throughput of 68.5% (Upfront Partitioning), 91.4% (Local Merge), and 91% (Global Merge) compared to memory bandwidth. For Local and Global Merge, these numbers are a strong indicator that they are bounded by memory bandwidth. We approximate the overhead for using a JVM compared to C++ to a factor of 15 (Upfront Partitioning), 16 (Local Merge), and 13 (Global Merge). We analyze the implementations in-depth in Section 5.2.2 and 5.2.3 to identify causes of these large differences in performance.

As shown in Figure 8, Local and Global Merge outperform the Upfront Partitioning strategy by a factor of 1.34 and 1.33. The main reason for the sub-optimal scaling of Upfront Partitioning is the use of queues. First, queues represent a bottleneck if their maximum throughput is reached (see Section 3.2). Second, queues require the compiler to generate two distinct code segments that are executed by different threads such that tuples cannot be passed via registers. Thus, queues lead to less efficient code, which we investigate in detail in Section 5.2.3. In contrast, Local and Global Merge perform similar for YSB and almost reach the physical memory limit of a single node. The main reason is that threads work independently from each other because the overall synchronization among threads is minimal.

LRB Benchmark. In Figure 9, we present throughput results for the LRB benchmark. Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 0.27% (Flink), 0.03% (Spark Streaming), and 0.01% (Storm). We approximate the overhead of each framework by comparing their throughput with the Java implementation of the upfront partitioning. Such overhead ranges from a factor of 2.7 (Flink), over a factor of 9 (Spark Streaming), to a factor of 39 (Storm). Note that, Spark Streaming processes data streams in micro-batches [90]. We observe that the scheduling overhead for many small micro-batch jobs negatively affects the overall throughput.

The scale-up optimized SPEs Streambox (written in C++) and Saber (written in Java) achieves about 9% and less than 1% of the theoretically possible throughput, respectively. In contrast, the C++ implementations achieve a throughput of 15% (UP), 48% (LM), and 67% (GM) of the physical limit. We approximate the overhead of a JVM compared to C++ to a factor of 20 (UP), 40 (LM), and 55 (GM).

Compared to YSB, the overall throughput of the LRB is lower because it is more complex, requires more synchronization, and larger operator states. In particular, the consumer requires lookups in multiple data structures compared to a single aggregation in the YSB. As a result, concurrent access by different threads to multiple data structures introduces cache thrashing and thus prevents the LRB implementations from achieving a throughput close to the physical limit.

Overall, the Global Merge strategy achieves the highest throughput because it writes in a shared data structure when windows end. In particular, we use a lock-free hash table based on atomic compare-and-swap instructions and open-addressing [51]. Thus, each processing thread writes its results for a range of disjoint keys, which leads to lower contention on the cells of the hash table. As a result, the C++ implementation of the Global Merge outperforms the Java implementation by a factor of 54 because it uses fine-grained atomic instructions (assembly instructions) that are not available in Java. In contrast, the C++ late Merge implementation is slower than Global Merge by a factor of 1.4. The main reason is that the additional merging step at the end of a window introduces extra overhead in terms of thread synchronization. Furthermore, the Java implementation of the Local Merge and Global Merge perform similarly because they implement the same atomic primitives.

The upfront partitioning is slower than Local Merge by a factor of 3.2 because it utilizes queues, buffers, and extra threads, which introduce extra overhead. An analysis of the execution reveals that the increased code complexity results in a larger code footprint. Furthermore, the upfront partitioning materializes input tuples on both sides of the queue. As a result, the Java implementation of upfront partitioning is slower than the C++ counterpart by a factor of 20.

In the remainder of this evaluation, we use Flink and Streambox as representative baselines for state-of-the-art scale-up and scale-out SPEs. Furthermore, we restrict the

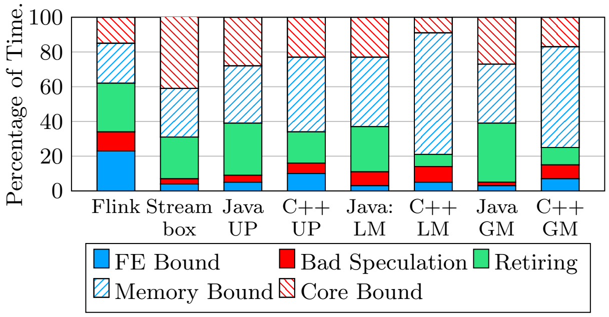


Figure 11: Execution time breakdown YSB.

in-depth performance analysis to the YSB because the overall results of YSB and LRB are similar.

New York Taxi Query In Figure 10, we present the throughput results for the NYT query. As shown, Flink, Spark Streaming, and Storm exploit only a small fraction of physical limit, i.e., 0.5% (Flink), 0.37% (Spark Streaming), and 0.34% (Storm). The pure Java implementations perform faster and exploit 16% (Java UP), 29% (Java LM), and 20% (Java GM) of the available memory bandwidth. In contrast, the C++ based implementation achieve a much higher bandwidth utilization of 69% (C++ UP), 80% (C++ LM), and 94% (C++ GM). Finally, the scale-out optimized SPEs Streambox (17%) and Saber (6%) perform better than the scale-out optimized SPEs. Compared to the YSB, this query induces simpler instructions but each tuple is larger, which leads to better performing pure-Java implementations. However, the the C++ implementations are utilizing the memory bandwidth more efficiently and the best variant (GM) achieves throughput values near the physical memory limit.

5.2.2 Execution Time Breakdown

In this section, we break down the execution time for different CPU components following the Intel optimization guide [41]. This time breakdown allows us to identify which parts of the micro-architecture are the bottleneck.

Metrics. Intel provides special counters to monitor buffers that feed micro-ops supplied by the front-end to the out-of-order back-end. Using these counters, we are able to derive which CPU component stalls the CPU pipeline and for how long. In the following, we describe these components in detail. First, the *front-end* delivers up to four micro-ops per cycle to the back-end. If the front-end stalls, the rename/allocate part of the out-of order engine starves and thus execution becomes front-end bound. Second, the *back-end* processes instructions issued by the front-end. If the back-end stalls because all processing resources are occupied, the execution becomes back-end bound. Furthermore, we divide back-end stalls into stalls related to the memory sub-system (called *Memory bound*) and stalls related to the execution units (called *Core bound*). Third, *bad speculation* summarizes the time that the pipeline executes speculative micro-ops that never successfully retire. This time represents the amount of work that is wasted by branch mispredictions. Fourth, *retiring* refers to the number of cycles that are actually used to execute useful instructions. This time represents the amount of useful work done by the CPU.

Experiment. In Figure 11, we breakdown the execution of cycles spent in the aforementioned five CPU components based on the Intel optimization guide [41].

Flink is significantly bounded by the front-end and bad speculation (up to 37%). Furthermore, it spends only a small portion of its cycles waiting on data (23%). This indicates that Flink is *CPU bound.* In contrast, Streambox is largely core bounded which indicates that the generated code does not use the available CPU resources efficiently.

Table 1: Resource utilization of design alternatives.

	Flink	Stream-box	Java: UP	C++: UP	Java: LM	C++: LM	Java: GM	C++: GM
Branches /Input Tuple	549	350	418	6.7	191	2.7	192	2.6
Branch Mispred./ Input Tuple	1.84	2.53	1.44	0.37	0.77	0.37	0.79	0.36
L1D Misses/ Input Tuple	52.16	42.4	12.86	1.93	5.66	1.41	7.31	1.42
L2D Misses/ Input Tuple	81.93	107.09	34.42	3.54	14.50	2.22	23.79	3.66
L3 Misses/ Input Tuple	37.26	72.04	13.97	1.67	6.88	1.56	6.70	1.83
TLBD Misses/ Input Tuple	0.225	0.825	0.145	0.00035	0.01085	0.000175	0.00292	0.0012
L1I Misses/ Input Tuple	19.5	1.35	0.125	0.00125	0.0155	0.0002	0.0229	0.000075
L2I Misses/ Input Tuple	2.2	0.6	0.075	0.001	0.008	0.0001	0.009	0.00007
TLBI Misses/ Input Tuple	0.04	0.3	0.003	0.0004	0.0007	0.000008	0.0009	0.000003
Instr. Exec./ Input Tuple	2,247	2,427	1,344	136	699	49	390	64

All Java implementations are less front-end bound and induce less bad speculation and more retiring instructions compared to their respective C++ implementations. In particular, the Java implementations spend only few cycles in the front-end or due to bad speculation (5%-11%). The remaining time is spent in the back-end (23%-28%) by waiting for functional units of the out-of-order engine. The main reason for that is that Java code accesses tuples using a random access pattern and thus spend the majority of time waiting for data. Therefore, an instruction related stall time is mainly overlapped by long lasting data transfers. In contrast, C++ implementations supply data much faster to the CPU such that the front-end related stall time becomes more predominant. We conclude that Java implementations are significantly *core bound*. LRB induces similar cycle distributions and is therefore not shown.

All C++ implementations are more memory bound (43%-70%) compared to their respective Java implementations (33%-40%). To further investigate the memory component, we analyze the exploited memory bandwidth. Flink and Streambox utilize only 20% of the available memory bandwidth. In contrast, C++ implementations utilize up to 90% of the memory bandwidth and Java implementations up to 65%. In particular, the Java implementations exploit roughly 70% of the memory bandwidth of their respective C++ implementations. Therefore, other bottlenecks like inefficient resource utilization inside the CPU may prevent a higher record throughput. Finally, the more complex code of LRB results in 20% lower memory bandwidth among all implementations (not shown here). We conclude that C++ implementations are *memory bound.*

Outlook. The memory bound C++ implementations can benefit from an increased bandwidth offered by future network and memory technologies. In contrast, Java implementations would benefit less from this trend because they are more core bound compared to the C++ implementations. In particular, a CPU becomes core bound if its computing resources are inefficiently utilized by resource starvation or non-optimal execution ports utilization. In the same way, state-of-the-art scale-out SPEs would benefit less from increased bandwidth because they are significantly front-end bound. In contrast, Streambox as a scale-up SPE would benefit from an improved code generation that utilizes the available resources more efficiently. Overall, Java and current SPEs would benefit from additional resources inside the CPU, e.g., register or compute units, and from code that utilizes the existing resources more efficiently.

5.2.3 Analysis of Resource Utilization

In this section, we analyze different implementations of YSB regarding their resource utilization. In Table 1, we present sampling results for Flink and Streambox as well as Upfront Partitioning (Part.), Local Merge (LM), and Global Merge (GM) implementations in Java and C++. Each implementation processes 10M records per thread and the sampling results reflect the pure execution of the workload without any preliminary setup.

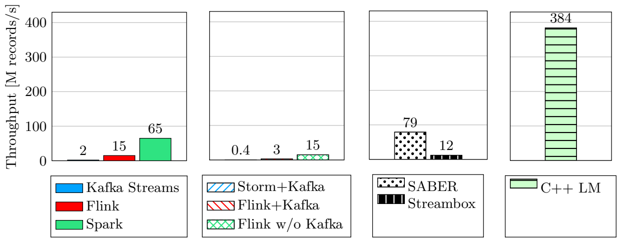
Control Flow. The first block of results presents the number of branches and branch mispredictions. These counters are used to evaluate the control flow of an implementation. As shown, the C++ implementations induce only few branches as well as branch mispredictions compared to the Java implementations (up to a factor of 3) and Flink (up to a factor of 5). However, all C++ implementations induce the same number of branch mispredictions but different number of branches. The main reason for that is that UP induces additional branches for looping over the buffer. Since loops induce only few mispredictions, the number of mispredictions does not increase significantly.

Overall, C++ implementations induce a lower branch misprediction rate that wastes fewer cycles for executing mispredicted instructions and loads less unused data. Furthermore, a low branch misprediction rate increases code locality. Regarding the state-of-the-art SPEs, Streambox induces less branches but more branch mispredictions.

Data Locality. The second block presents data cache related counters which can be used to evaluate the data locality of an implementation. Note that, YSB induces no tuple reuse because a tuple is processed once and is never reloaded. Therefore, data-related cache misses are rather high for all implementations. As shown, the C++ implementations outperform the Java based implementations by inducing the least amount of data cache misses per input tuple in all three cache levels. The main reason is the random memory access pattern that is introduced by object scattering in memory for JVM-based implementations. Furthermore, all C++ implementations produce similar numbers but the Java implementations differ significantly. Interestingly, the Java Upfront Partitioning implementation and Flink induce a significantly higher number of misses in the data TLB cache, which correlates directly to their low throughput (see Figure 8). Additionally, Streambox induces the most data related cache and TLB misses which indicates that they utilize a sub-optimal data layout.

Overall, C++ implementations induce up to 9 times less cache misses per input tuple which leads to a higher data locality. This higher data locality results in shorter access latencies for tuples and thus speeds up execution significantly.

Code Locality. The third block presents instruction cache related counters, which can be used to evaluate the code locality of an implementation. The numbers show that the C++ implementations create more efficient code that exhibits a high instruction locality with just a few instruction cache misses. The high instruction locality originates from a small instruction footprint and only few mispredicted branches. This indicates that the instructions footprint of the C++ implementations of YSB fits into the instruction cache of a modern CPU. Furthermore, the late merging strategies produce significantly fewer cache misses compared to the Upfront Partitioning strategy. The misses in the instruction TLB follow this behavior.



(a) 10 Nodes [49] (b) 10 Nodes [18] (c) 1 Node [88] (d) optimized

Figure 12: YSB reported throughput.

The last block presents the number of executed instructions per input tuple which also impacts the code locality. Again, the C++ implementations execute significantly fewer instructions compared to Java implementations. Furthermore, the late merging strategies execute fewer instructions than the UP. This is also reflected in the throughput numbers in Figure 8. Since the C++ implementations induce fewer instruction cache misses compared to the respective Java implementations, we conclude a larger code footprint and the virtual function calls of a JVM are responsible for that. Interestingly, Flink executes almost as many instructions as Streambox but achieves a much lower throughput.

For all instruction related cache counters, Flink induces significantly more instruction cache misses compared to Java implementations by up to three orders of magnitude. The main reason for this is that the code footprint of the generated UDF code exceeds the size of the instruction cache. In contrast, Streambox generates smaller code footprints and thus induces less instruction related cache misses.

Summary. All C++ implementations induce a better control flow, as well as a higher data and instruction locality. In contrast, Java-based implementations suffer from JVM overheads such as pointer chasing and transparent memory management. Overall, both late merging strategies outperform Upfront Partitioning. Finally, Flink and Streambox are not able to exploit modern CPUs efficiently.

5.2.4 Comparison to Cluster Execution

In this section, we compare our hand-written implementations with scale-out optimized SPEs on a cluster of computing nodes. Additionally, we present published results from the scale-up SPEs SABER and Streambox. In Figure 12, we show published benchmarking results of YSB on a cluster [49, 18, 88]. We choose to use external numbers because the vendors carefully tuned their systems to run the benchmark as efficiently as possible on their cluster.

The first three numbers are reported by databricks [18] on a cluster of 10 nodes. They show that *Structured Streaming*, an upcoming version of Apache Spark, significantly improves the throughput (65M rec/sec) compared to Kafka Streams (2M rec/sec) and Flink (15M rec/sec). The next three numbers are reported by dataArtisans [49] on a cluster of 10 nodes. They show that Storm with Kafka achieves only 400K recs/sec and Flink using Kafka achieves 3M rec/second. However, if they omit Kafka as the main bottleneck and move data generation directly into Flink, they increase throughput up to 15M recs/sec.

Figure 12 shows, even when using 10 compute nodes, the throughput is significantly below the best performing single node execution that we present in this paper (C++ LM). The major reason for this inefficient scale-out is the partitioning step involved in distributed processing. As the number of nodes increases, the network traffic also increases.

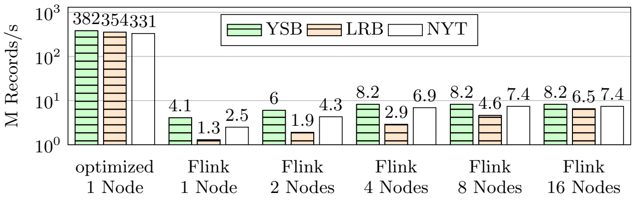


Figure 13: Scale-out experiment.

Therefore, the network bandwidth becomes the bottleneck and limits the overall throughput of the distributed execution. However, even without the limitations on network bandwidth by using one node and main memory, state-of-the-art SPEs cannot utilize modern CPUs efficiently (see Figure 8). One major reason for this is that their optimization decisions are focused on scale-out instead of scale-up.

The next two benchmarking results are reported in [73] which measure the throughput of SABER and Streambox on one node. Although they achieve a higher throughput compared to scale-out SPEs, they still do not exploit the capabilities of modern CPUs entirely. Note that the numbers reported in [73] differ from our numbers because we use different hardware. Furthermore, we change the implementation of YSB on Streambox to process characters instead of strings and optimize the parameter configuration such that we can improve the throughput to 71M rec/sec (see Figure 8). The last benchmarking result reports the best performing hand-coded C++ implementation presented in this paper which achieves a throughput of up to 384M rec/sec.

As a second experiment, we demonstrate the scale-out performance of Flink on the YSB, LRB, and NYT and compare it to a single node implementation. Note that, we did the same scale-out experiments for all queries on Spark and Storm and they show the same behaviour; therefore, we omit them in Figure 13. We execute the benchmarks on up to 16 nodes, whereas each node contains a 16-core Intel Xeon CPU (E5620 2.40GHz) and all nodes are connected via 1Gb Ethernet. In Figure 13, we compare our single node optimized version (DOP=1) with the the scale-out execution (DOP 2-16). As shown, YSB scales to up four nodes with a maximum throughput of 8.2M recs/sec. In contrast, NYT scales to up eight nodes with a maximum throughput of 7.4M recs/sec. The main reason for this sub-optimal scaling is that the benchmarks are network-bound. Flink generates the input data inside the engine and uses its state management APIs to save the intermediate state. However, the induced partitioning utilizes the available network bandwidth entirely. In contrast, LRB does increase the throughput for each additional node but the overall increase is only minor.

Overall, Figure 13 reveals that even if we scale-out to a large cluster of nodes, we are not able to provide the same throughput as a hand-tuned single node solution. However, a recent study shows that as the network bandwidth increases, the data processing for distributed engines becomes CPU-bound [84]. The authors conclude that current data processing systems, including Apache Spark and Apache Flink need systematic changes.

5.2.5 RDMA

In this experiment, we want to examine if hand-written Java or C++ implementations are capable of utilizing the ever rising available network bandwidth in the future. To this end, we extend the YSB Benchmark such that one node produces the data, as it will be the case in the streaming scenario, and one node gets the stream of data via Infiniband and RDMA. In Figure 14, we show the throughput of our hand-written implementation of the YSB using Java and C++ on an E7-4850v4 CPU and an InfiniBand FDR network. With *physical limit*, we refer to the maximum bandwidth following the specification of 56 Gbit/s (FDR 4x). For Java as well as for C++, we implement one version that just sends data via the network without processing it, i.e., read only, and one version that includes the processing of the YSB, i.e., with proc. As shown, a C++ implementation that sends data over a high-speed InfiniBand network and processes it is able to reach nearly the same throughput as the implementation without processing. In contrast, the Java implementation with processing achieves only 22% of the throughput without processing. This indicates that the Java implementation is bounded by the processing instead of the network transfer. This is in line with the results presented in the previous sections. Overall, the C++ implementation is able to exploit 70% of the theoretical bandwidth without processing compared to 38% using Java.

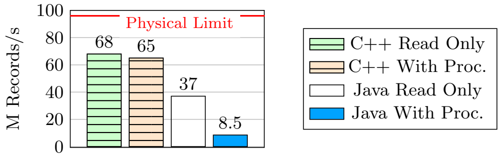


Figure 14: Infiniband exploitation using RDMA.

5.2.6 Latency

In this experiment, we compare the latencies of the Upfront Partitioning and the Global Merge parallelization strategy for a C++ implementations processing and 10M YSB input tuples. With latency, we refer to the time span between the end of a window and the materialization of the window at the sink. Our results show, that the Global Merge exhibits an average latency of 57ns (min: 38ns; max: 149ns). In contrast, Upfront Partitioning induces a one order of magnitude higher latency of 116μs (min: 112μs; max: 122μs). The main reason for this is the delay introduced by queues and buffers.

In all our experiments, Global Merge implementations achieve very low latencies in the order of a hundred nanoseconds. This is in contrast to state-of-the-art streaming systems which use Upfront Partitioning [29, 81, 91]. In particular, these systems have higher latencies (in the order of ms) for the YSB [39, 49, 88]. The Linear Road Benchmark defines a fixed latency upper bound of 5 seconds which our implementation satisfies by far.

5.3 Discussion

Our work is driven by the key observation that emerging network technologies allow us to ingest data with main memory bandwidth on a single scale-up server [22]. We investigate design aspects of an SPE optimized for scale-up. The results of our extensive experimental analysis are the following key insights:

Avoid high level languages. Implementing the critical code path or the performance critical data structures in Java results in a larger portion of random memory accesses and virtual function calls. These issues slow down Java implementations up by a factor of 54 compared to our C++ implementation. Furthermore, our profiling results provide strong indication that current SPEs are CPU-bound.

Avoid queuing and apply operator fusion. An SPE using queues as a mean to exchange data between operators is not able to fully exploit the memory bandwidth. Thus, data exchange should be replaced by operator fusion, which is enabled by query compilation techniques [70].

Use Late Merge parallelization. The parallelization based on Upfront Partitioning introduces an overhead by up to a factor of 4.5 compared to late merging that is based on logically dividing the stream in blocks. The key factor that makes working on blocks applicable in an SPE is that the network layer performs batching on a physical level already.

Use lock-free windowing. Our experiments reveal that a lock-free windowing implementation achieves a high throughput for a SPE on modern hardware. Furthermore, our approach enables queue-less late merging.

Scale-Up is an alternative. In our experiments, a single node using an optimized C++ implementation outperforms a 10 node cluster running state-of-the-art streaming systems. We conclude that scale-up is a viable way of achieving high throughput and low latency stream processing.

6. RELATED WORK

In this section, we cover additional related work that we did not discuss already. We group related work by topics.

Stream Processing Engines (SPEs). The first generation of streaming systems has built the foundation for today's state-of-the-art SPEs [12, 11, 20, 35, 36]. While the first generation of SPEs explores the design space of stream processing in general, state-of-the-art systems focus on throughput and latency optimizations for high velocity data streams [13, 21, 29, 32, 69, 81, 86, 91]. Apache Flink [29], Apache Spark [91], and Apache Storm [81] are the most popular and mature open-source SPEs. These SPEs optimize the execution to *scaling-out* on shared-nothing architectures. In contrast, another line of research focuses on *scaling-up* the execution on a single machine. Their goal is to exploit the capabilities of one high-end machine efficiently. Recently proposed SPEs in this category are Streambox [67], Trill [34], or Saber [58]. In this paper, we examine the data-related and processing-related design space of scale-up and scale-out SPEs in regards to the exploitation of modern hardware. We showcase design changes that are applicable in many SPEs to achieve higher throughput on current and future hardware technologies. In particular, we lay the foundation by providing building blocks for a third generation SPE, which runs on modern hardware efficiently.

Data Processing on Modern Hardware. In-memory databases explore fast data processing near memory bandwidth speed [23, 27, 70, 44, 95]. Although in-memory databases support fast state access [57, 26], their overall stream processing capabilities are limited. In this paper, we widen the scope of modern hardware exploitation to more complex stream processing applications such as LRB, YSB, and NYT. In this field, SABER also processes complex streaming workloads on modern hardware [58]. However, it combines a Java-based SPE with GPGPU acceleration. In contrast, we focus on outlining the disadvantages of a Java-based SPEs. Nevertheless, the design changes proposed in this paper are partially applicable for JVM-based SPEs.

Performance Analysis of SPE. Zhang et al. [94] deeply analyze the performance characteristics of Storm and Flink. They contribute a NUMA-aware scheduler as well as a system-agnostic not-blocking tuple buffering technique. Complementary to the work of Zhang, we investigate fundamental design alternatives for SPEs, specifically data passing, processing models, parallelization strategies, and an efficient windowing technique. Furthermore, we consider upcoming networking technologies and existing scale-up systems.

Ousterhout et al. [71] analyze Spark and reveal that many workloads are CPU-bound and not disk-bound. In contrast, our work focuses on understanding why existing engines are CPU-bound and on systematically exploring means to overcome these bottlenecks by applying scale-up optimizations.

Trivedi et al. assess the performance-wise importance of the network for modern distributed data processing systems such as Spark and Flink [84]. They analyze query runtime and profile the systems regarding the time spent in each major function call. In contrast, we perform a more fine-grained analysis of hardware usage of those and other SPEs and native implementations.

Query Processing and Compilation. In this paper, we used hand-coded queries to show potential performance advantages for SPEs using query compilation. Using established code generation techniques proposed by Krikellas [59] or Neumann [70] allows us to build systems that generate optimized code for any query, i.e., also streaming queries. However, as opposed to a batch-style system, input tuples are not immediately available in streaming systems. Furthermore, the unique requirements of windowing semantics introduce new challenges for a query compiler. Similarly to Spark, we adopt code generation to increase throughput. In contrast, we directly generate efficient machine code instead of byte code, which requires an additional interpretation step for the execution.

7. CONCLUSION

In this paper, we analyze the design space of SPEs optimized for scale-up. Our experimental analysis reveals that the design of current SPEs cannot exploit the full memory bandwidth and computational power of modern hardware. In particular, SPEs written in Java cause a larger number of random memory accesses and virtual function calls compared to C++ implementations.

Our hand-written implementations that use appropriate streaming optimizations for modern hardware achieve near memory bandwidth and outperform existing SPEs in terms of throughput by up to two orders of magnitude on three common streaming benchmarks. Furthermore, we show that carefully-tuned single node implementations outperform existing SPEs even if they run on a cluster. Emerging network technologies can deliver similar or even higher bandwidth compared to main memory. Current systems cannot fully utilize the current memory bandwidth and, thus, also cannot exploit faster networks in the future.

With this paper, we lay the foundation for a scale-up SPE by exploring design alternatives in-depth and describing a set of design changes that can be incorporated into current and future SPEs. In particular, we show that a queue-less execution engine based on query compilation, which eliminates tuple queues as a mean to exchange intermediate results between operators, is highly suitable for an SPE on modern hardware. Furthermore, we suggest to replace Upfront Partitioning with Late Merging strategies. Overall, we conclude that with emerging hardware, scale-up is a viable alternative to scale-out.

Acknowledgments. This work was funded by the EU projects E2Data (780245), DFG Priority Program "Scalable Data Management for Future Hardware" (MA4662-5), and the German Ministry for Education and Research as BBDC I (01IS14013A) and BBDC II (01IS18025A).

8. REFERENCES

[1] Apache Storm Trident. *URL storm. apache. org/ releases/ 1. 1. 1/ Trident-tutorial. html.*

[2] Disni: Direct storage and networking interface, 2017. Retrieved March 30, 2018, from `https://github.com/zrlio/disni`.

[3] I. t. association. infiniband roadmap, 2017. Retrieved October 30, 2017, from `http://www.infinibandta.org/`.

[4] Package java.util.concurrent, 2017. Retrieved October 30, 2017, from `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html`.

[5] A ReaderWriter queue which shows superior performance in benchmarks, 2017. Retrieved October 30, 2017, from `https://github.com/cameron314/readerwriterqueue/tree/master/benchmarks`.

[6] Sparkrdma shufflemanager plugin, 2017. Retrieved March 30, 2018, from `https://github.com/Mellanox/SparkRDMA`.

[7] A SPSC implementation from Facebook, 2017. Retrieved October 30, 2017, from `https://github.com/facebook/folly/tree/master/folly`.

[8] A SPSC queue implemented by the Boost library, 2017. Retrieved October 30, 2017, from `www.boost.org/doc/libs/1\_64\_0/doc/html/boost/lockfree/queue.html`.

[9] A SPSC queue implemented by the TBB library, 2017. Retrieved October 30, 2017, from `https://www.threadingbuildingblocks.org/docs/doxygen/a00035.html`.

[10] Hotspot virtual machine garbage collection tuning guide, 2018. Retrieved November 2, 2018, from `https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html`.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al. The design of the borealis stream processing engine. In *CIDR*, volume 5, pages 277–289, 2005.

[12] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[13] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: fault-tolerant stream processing at internet scale. *PVLDB*, 6(11):1033–1044, 2013.

[14] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *PVLDB*, 8(12):1792–1803, 2015.

[15] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[16] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts. Linear road: a stream data management benchmark. In *VLDB*, pages 480–491. 2004.

[17] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347. VLDB Endowment, 2004.

[18] M. Armbrust. Making apache spark the fastest open source streaming engine, 2017. Databricks Engineering Blog, URL https://databricks.com/blog/2017/06/06/simple-super-fast-streaming-engine-apache-spark.html.

[19] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke. Nephele/PACTs: A programming model and execution framework for web-scale analytical processing. In *SoCC*, pages 119–130. ACM, 2010.

[20] T. Bernhardt and A. Vasseur. Esper: Event stream processing and correlation, 2007.

[21] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM infosphere streams for scalable, real-time, intelligent transportation services. In *SIGMOD*, pages 1093–1104. ACM, 2010.

[22] C. Binnig, A. Crotty, A. Galakatos, T. Kraska, and E. Zamanian. The end of slow networks: It's time for a redesign. *PVLDB*, 9(7):528–539, 2016.

[23] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[24] I. Botan, Y. Cho, R. Derakhshan, N. Dindar, L. Haas, K. Kim, C. Lee, G. Mundada, M.-C. Shan, N. Tatbul, et al. Design and implementation of the maxstream federated stream processing architecture. 2009.

[25] I. Botan, D. Kossmann, P. M. Fischer, T. Kraska, D. Florescu, and R. Tamosevicius. Extending xquery with window functions. In *VLDB*, pages 75–86. 2007.

[26] L. Braun, T. Etter, G. Gasparis, M. Kaufmann, D. Kossmann, D. Widmer, A. Avitzur, A. Iliopoulos, E. Levy, and N. Liang. Analytics in motion: High performance event-processing and real-time analytics in the same database. In *SIGMOD*, pages 251–264. ACM, 2015.

[27] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *SIGMOD*, pages 1891–1906. ACM, 2016.

[28] C. Cameron, J. Singer, and D. Vengerov. The judgment of forseti: Economic utility for dynamic heap sizing of multiple runtimes. In *ACM SIGPLAN Notices*, volume 50, pages 143–156. ACM, 2015.

[29] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[30] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Engineering Bulletin*, 36(4), 2015.

[31] P. Carbone, J. Traub, A. Katsifodimos, S. Haridi, and V. Markl. Cutty: Aggregate sharing for user-defined windows. In *CIKM*, pages 1201–1210. 2016.

[32] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *SIGMOD*, pages 725–736. 2013.

[33] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. Henry, R. Bradshaw, and Nathan. Flumejava: Easy, efficient data-parallel pipelines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375. 2 Penn Plaza, Suite 701 New York, NY 10121-0701, 2010.

[34] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing. Trill: A high-performance incremental query processor for diverse analytics. *Proc. VLDB Endow.*, 8(4):401–412, 2014.

[35] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: continuous dataflow processing. In *SIGMOD*, pages 668–668. ACM, 2003.

[36] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *ACM SIGMOD Record*, volume 29, pages 379–390. ACM, 2000.

[37] Q. Chen, M. Hsu, and H. Zeller. Experience in continuous analytics as a service (CaaaS). In *EDBT*, pages 509–514. 2011.

[38] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, et al. Benchmarking streaming computation engines: Storm, Flink and Spark streaming. In *International Parallel and Distributed Processing Symposium, Workshops*, pages 1789–1792. 2016.

[39] S. Chintapalli, D. Dagit, B. Evans, R. Farivar, T. Graves, M. Holderbaugh, Z. Liu, K. Nusbaum, K. Patil, B. J. Peng, and P. Poulosky. Benchmarking streaming computation engines at yahoo, 2015.

[40] R. Consortium. *RDMA Protocol Verbs Specification (Version 1.0)*. April 2003.

[41] I. Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Number 325462-044US. August 2012.

[42] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 10–10. USENIX Association, 2004.

[43] S. Ewen. Off-heap memory in apache flink and the curious jit compiler, 2015. Retrieved November 2, 2018, from `https://flink.apache.org/news/2015/09/16/off-heap-memory.html`.

[44] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. Sap hana database: data management for modern business applications. *ACM Sigmod Record*, 40(4):45–51, 2012.

[45] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Making state explicit for imperative big data processing. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 49–60, Philadelphia, PA, 2014. USENIX Association.

[46] P. M. Fischer, K. S. Esmaili, and R. J. Miller. Stream schema: providing and exploiting static metadata for data stream processing. In *EDBT*, pages 207–218. 2010.

[47] B. Gedik. Generic windowing support for extensible stream processing systems. *Software: Practice and Experience*, 44(9):1105–1128, 2014.

[48] B. Gedik, H. Andrade, and K. Wu. A code generation approach to optimizing high-performance distributed data stream processing. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management, CIKM 2009, Hong Kong, China, November 2-6, 2009*, pages 847–856, 2009.

[49] J. Grier. Extending the yahoo! streaming benchmark, 2016.

[50] M. Grossniklaus, D. Maier, J. Miller, S. Moorthy, and K. Tufte. Frames: data-driven windows. In *DEBS*, pages 13–24. 2016.

[51] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *International Symposium on Distributed Computing*, pages 350–364. Springer, 2008.

[52] M. Hirzel, H. Andrade, B. Gedik, G. Jacques-Silva, R. Khandekar, V. Kumar, M. Mendell, H. Nasgaard, S. Schneider, R. Soulé, et al. IBM streams processing language: Analyzing big data in motion. *IBM Journal of Research and Development*, 57(3/4):7–1, 2013.

[53] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, 2014.

[54] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road bnchmark on the stream processing core. In *SIGMOD*, pages 431–442. 2006.

[55] C. S. Jensen, C. M. Jermaine, and X. Zhou, editors. *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*. IEEE Computer Society, 2013.

[56] A. Kalia, M. Kaminsky, and D. G. Andersen. Design guidelines for high performance rdma systems. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '16, pages 437–450, Berkeley, CA, USA, 2016. USENIX Association.

[57] A. Kipf, V. Pandey, J. Böttcher, L. Braun, T. Neumann, and A. Kemper. Analytics on fast data: Main-memory database systems versus modern streaming systems. In *EDBT*, pages 49–60. 2017.

[58] A. Koliousis, M. Weidlich, R. Castro Fernandez, A. L. Wolf, P. Costa, and P. Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *SIGMOD*, pages 555–569. ACM, 2016.

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

[60] S. Krishnamurthy, M. J. Franklin, J. Davis, D. Farina, P. Golovko, A. Li, and N. Thombre. Continuous analytics over discontinuous streams. In *SIGMOD*, pages 1081–1092. 2010.

[61] S. Krishnamurthy, C. Wu, and M. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD*, pages 623–634. 2006.

[62] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754. ACM, 2014.

[63] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. The art of practical synchronization. In *DaMoN*, page 3. ACM, 2016.

[64] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD Record*, 34(1):39–44, 2005.

[65] E. Liarou, R. Goncalves, and S. Idreos. Exploiting the power of relational databases for efficient stream processing. In *EDBT*, pages 323–334. 2009.

[66] D. Lion, A. Chiu, H. Sun, X. Zhuang, N. Grcevski, and D. Yuan. Don't get caught in the cold, warm-up your jvm: Understand and eliminate jvm warm-up overhead in data-parallel systems. In *OSDI*, pages 383–400, 2016.

[67] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin. Streambox: Modern stream processing on a multicore machine. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 617–629, Santa Clara, CA, 2017. USENIX Association.

[68] C. Mitchell, Y. Geng, and J. Li. Using one-sided rdma reads to build a fast, cpu-efficient key-value store. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 103–114, Berkeley, CA, USA, 2013. USENIX Association.

[69] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *SOSP*, pages 439–455. ACM, 2013.

[70] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[71] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *NSDI*, pages 293–307. USENIX Association, 2015.

[72] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *PVLDB*, 3(1-2):928–939, 2010.

[73] P. Pietzuch, P. Garefalakis, A. Koliousis, H. Pirk, and G. Theodorakis. Do we need distributed stream processing?, 2018.

[74] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *Proc. VLDB Endow.*, 8(12):1442–1453, 2015.

[75] W. Rödiger, T. Mühlbauer, A. Kemper, and T. Neumann. High-speed query processing over high-speed networks. *Proc. VLDB Endow.*, 9(4):228–239, 2015.

[76] T. Schneider. Analyzing 1.1 billion nyc taxi and uber trips, with a vengeance, 2015.

[77] M. Svensson. *Benchmarking the performance of a data stream management system.* PhD thesis, MSc thesis report, Uppsala University, 2007.

[78] K. Tangwongsan, M. Hirzel, and S. Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 66–77, New York, NY, USA, 2017. ACM.

[79] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu. General incremental sliding-window aggregation. *PVLDB*, 8(7):702–713, 2015.

[80] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@ twitter. In *SIGMOD*, pages 147–156. ACM, 2014.

[81] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, et al. Storm@Twitter. In *SIGMOD*, pages 147–156. 2014.

[82] J. Traub, P. Grulich, A. R. Cuéllar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Efficient window aggregation with general stream slicing. In *22nd International Conference on Extending Database Technology (EDBT)*, 2019.

[83] J. Traub, P. M. Grulich, A. R. Cuellar, S. Breß, A. Katsifodimos, T. Rabl, and V. Markl. Scotty: Efficient window aggregation for out-of-order stream processing. In *34th International Conference on Data Engineering (ICDE)*, pages 1300–1303. IEEE, 2018.

[84] A. Trivedi, P. Stuedi, J. Pfefferle, R. Stoica, B. Metzler, I. Koltsidas, and N. Ioannou. On the [ir]relevance of network performance for data processing. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, 2016.

[85] D. Vyukov. Single-producer/single-consumer queue. *Intel Developer Zonw, URL software.intel.com/en-us/articles/single-producer-single-consumer-queue*, 2015.

[86] Y. Wu and K.-L. Tan. Chronostream: Elastic stateful stream computation in the cloud. In *ICDE*, pages 723–734. IEEE, 2015.

[87] R. Xin and J. Rosen. Project tungsten: Bringing apache spark closer to bare metal, 2015. Retrieved November 2, 2018, from URL https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html.

[88] B. Yavuz. Benchmarking structured streaming on databricks runtime against state-of-the-art streaming systems, 2017.

[89] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. 2012.

[90] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *SOSP*, pages 423–438. ACM, 2013.

[91] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[92] E. Zeitler and T. Risch. Scalable splitting of massive data streams. In *DASFAA*, pages 184–198. 2010.

[93] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.

[94] S. Zhang, B. He, D. Dahlmeier, A. C. Zhou, and T. Heinze. Revisiting the design of data stream processing systems on multi-core processors. In *ICDE*, pages 659–670. 2017.

[95] M. Zukowski, P. A. Boncz, et al. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 2012.