Viper: Communication-Layer Determinism and Scaling in Low-Latency Stream Processing

Ivan Walulya(✉), Yiannis Nikolakopoulos, Vincenzo Gulisano, Marina Papatriantafilou, and Philippas Tsigas

Chalmers University of Technology, Gothenburg, Sweden
{walulya,ioaniko,vinmas,ptrianta,tsigas}@chalmers.se

Abstract. Stream Processing Engines (SPEs) process continuous streams of data and produce up-to-date results in a real-time fashion, typically through one-at-a-time tuple analysis. When looking into the vital SPE processing properties required from applications, determinism has a strong position besides scalability in throughput and low processing latency. SPEs scale in throughput and latency by relying on shared-nothing parallelism, deploying multiple copies of each operator to which tuples are distributed based on the semantics of the operator. The coordination of the asynchronous analysis of parallel operators required to enforce determinism is then carried out by additional dedicated sorting operators. In this work we shift such costly coordination to the communication layer of the SPE. Specifically, we extend earlier work on shared-memory implementations of deterministic operators and provide a communication module (Viper) which can be integrated in the SPE communication layer. Using Apache Storm and the Linear Road benchmark, we show the benefits that can be achieved by our approach in terms of throughput and energy efficiency of SPEs implementing one-at-a-time analysis.

Keywords: Data streaming · Low-latency
Shared-nothing and shared-memory parallelism
Stream processing engines

1 Introduction

Data streaming emerged to meet the stringent demands of massive on-line data analysis in a variety of contexts, such as cloud and edge-computing architectures. Stream Processing Engines (SPEs) allow programmers to formulate continuous queries, defined as Directed Acyclic Graphs of interconnected operators, that process incoming data producing results on a continuous fashion. Examples of such Stream Processing Engines include StreamCloud [12], Apache Storm [26], Apache Flink [10] and Saber [19].

Parallelism is key for modern hardware to achieve *high-throughput* and *low latency* in SPEs processing increasingly large data volumes in evolving cyber-physical infrastructures [16]. The importance of scaling in throughput and keeping low-latency processing in SPEs is clear, manifested also by work in elasticity

D. B. Heras and L. Bougé (Eds.): Euro-Par 2017 Workshops, LNCS 10659, pp. 129–140, 2018.
https://doi.org/10.1007/978-3-319-75178-8_11

of parallelism, e.g. [9,12]. With parallelization, though, careful orchestration of operators' execution is required to preserve *determinism*. An operator's implementation is *deterministic* if, given the same sequences of input tuples, the same sequence of output tuples is produced independently of the tuples' inter-arrival times or the degree of parallelism of the operator [14,15].

The guarantee of determinism in SPEs, under concurrent execution of parallel operators, relies on dedicated sorting operators that are either added to continuous queries by dedicated query compilers [12] or in SPEs such as Apache Storm [26], or are left to the application developers to place them within their streaming applications. Minimizing the computational overhead introduced by such dedicated operators (we refer to this as *operator-layer* determinism) is nevertheless challenging, especially for one-at-a-time, fine-grained low latency tuple processing. We address the issue of guaranteeing determinism in a modular, automated and efficient way. We start from the observation that, commonly in SPEs, each physical stream is piped from a producer (e.g., an incoming link from a sensor, or an outgoing link of an operator instance) to its consumer (another operator instance), without coordination or sharing state. Sharing and synchronizing efficiently in an automated way is the challenging key to provide a transparent determinism method to application developers, alleviating them from the responsibility of developing custom solutions and proof argumentation as required.

ScaleGate [15] is a data structure introduced for aggregate and join operators to guarantee determinism in a customized way. The work in this paper builds upon it and provides the following contributions: (i) It modularly shifts the procedure of guaranteeing determinism, from the operator-layer to the *communication layer* of an SPE, thus relieving application developers from the burden of devising application-dependent methods. (ii) It designs and implements a module, called *Viper*, which can be transparently integrated in an SPE communication layer. Building on ScaleGate, it lifts the data-structure's context into the communication layer of an SPE architecture. From ScaleGate to Viper, the novelty is on the transparency provided to the application developer in efficiently guaranteeing determinism. (iii) It integrates the module in Apache Storm (as a representative example of SPEs) and demonstrates via an extensive evaluation the feasibility of the idea of modularly providing determinism, while caring for efficiency in parallelism. The experimental evaluation of the proposed methodology used the Linear Road benchmark and shows the throughput as well as energy efficiency benefits, the latter being important with respect to sustainability of the evolution of processing infrastructures for cyberphysical systems.

The rest of the paper is organized as follows. We present preliminary concepts in Sect. 2. We describe our proposal for distinguishing the operator layer and communication layer in an SPE and discuss the advantages of doing that, we also introduce the Viper module, in Sect. 3. We evaluate the benefits of the Viper module in Sect. 4. Discuss related work and conclude in Sects. 5 and 6.

2 System Model

A stream is defined as an unbounded sequence of tuples $t_0, t_1, \ldots$ sharing the same schema composed of attributes $\langle ts,\ A_1, \ldots, A_n \rangle$. Given a tuple t, $t.ts$ represents its creation timestamp while $A_1, \ldots, A_n$ are application-related attributes.

Continuous queries (or simply queries in the remainder) are defined as DAGs of *operators* that consume and produce tuples. Operators are distinguished into *stateless* or *stateful*, depending on whether they keep any state that evolves with the tuples being processed. Stateless operators include Map (to alter the schema of tuples) and Filter (to discard or route tuples). Stateful operators include Aggregate (to compute aggregation functions such as sum or average over tuples) and Join (to match tuples coming from multiple streams). Due to the unbounded nature of streams, stateful operations are computed over *sliding windows*. Following the data streaming literature (e.g., [5,12,18]), we assume that streams fed by each data source contain timestamp-sorted tuples.

The performance of an operator depends on its *cost* and *selectivity*. That is, the average time needed to process an input tuple and (optionally) produce any resulting output tuple and the average number of output tuples produced upon the processing of one input tuple (e.g., an operator with selectivity 0.5 will produce, on average, one output tuple each time it processes two input tuples).

To illustrate the aforementioned terms and notions, Fig. 1A presents a sample streaming query from the Linear Road benchmark [4]. In this example, position reports are forwarded by vehicles traveling on a highway. The query checks if the report refers to a vehicle entering, leaving or changing a segment. In the affirmative case, it updates the number of vehicles and the tolls of the involved segments. Finally, it notifies the interested vehicles. The schema of each stream is presented on top of the operators. Aggregate A1 enriches each position report with the previous segment observed for the same vehicle. Subsequently, Filter F discards reports referring to vehicles that have not changed segment. Aggregate A2 updates the count for each segment. Finally, Map M computes the toll for a segment based on the number of vehicles in it and notifies vehicles.

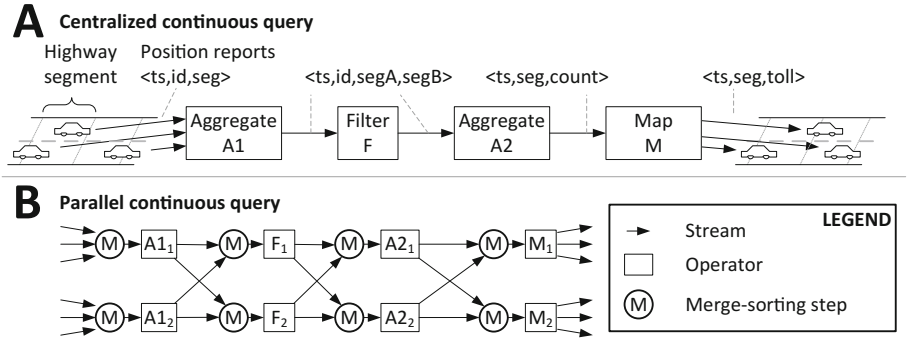


Fig. 1. Sample centralized and parallel query (Linear Road benchmark [4]).

2.1 Parallel and Deterministic Execution of Queries

A parallel version of a query, e.g. Fig. 1, is desirable to cope with large and fluctuating volume of tuples. Deterministic execution ensures that the results produced by the parallel query are exactly the same produced by its centralized counterpart. As explained in [12,13], determinism is enforced if the processing of each operator composing the query is deterministic. For an operator's processing to be deterministic, special merge-sorting steps[1] are defined before each operator *instance*, as shown in Fig. 1B, presenting a parallel version of the centralized query with two instances for each operator. The M steps merge-sort deterministically the incoming timestamp-sorted input streams of an operator instance into a single timestamp-sorted stream of tuples, allowing the operator instance's execution to be deterministic independently of the arrival interleaving of its input streams [12] by forwarding tuples when the latter are *ready*. Formally:

Definition 1 (ready tuple [14,15]**).** *Let t_i^j be the i-th tuple from timestamp-sorted stream S_j. t_i^j is* ***ready*** *to be processed if $t_i^j.ts \leq merge_{ts}$, where $merge_{ts} = min_k\{t_l^k.ts\}$ is the minimum timestamp among the timestamps in the set of tuples comprising the latest received tuples t_l^k from each timestamp-sorted stream S_k.*

2.2 Performance Metrics

We consider metrics that are commonly used to assess the performance of a streaming framework (from individual operators to queries or SPEs as a whole). More concretely, we take into account *throughput* and *latency* [12,15], as well as *energy consumption* [3]. Throughput, commonly measured in tuples per second (t/s), represents the maximum rate at which tuples can be fed to the operators composing a given query. Latency, commonly measured in milliseconds, represents the interleaving time between the forwarding of an output tuple and the timestamp carried by the latest input tuple contributing to it. For the energy consumption, we utilize RAPL energy counters [8] to measure power consumption in Watts and take the average over the counter samples during an execution.

3 From Operator- to Communication-Layer Determinism

As we explained in Sect. 1, determinism is typically enforced by SPEs at the operator layer. That is, the merge-sorting required to enforce determinism (cf. Sect. 2) is run by dedicated operators that are deployed together with the operators defined by the application programmer. Alternatively, as we propose and explain in this section, determinism can be achieved by the communication layer of an SPE, used for buffering operators' input and output tuples.

[1] We use the term steps rather than operators because, as shown in the following sections, merge-sorting and routing can be both assigned to dedicated operators or integrated in the communication layer of an SPE.

To introduce *layering* for SPE functionality provisioning, without loss of generality, we consider in the following the node shown in Fig. 2. The node depicts the operators F, $A2$ and M of Fig. 1B. Our discussion holds independently of whether other operators are deployed within the SPE running the query and of whether more than two instances are defined for each operator.

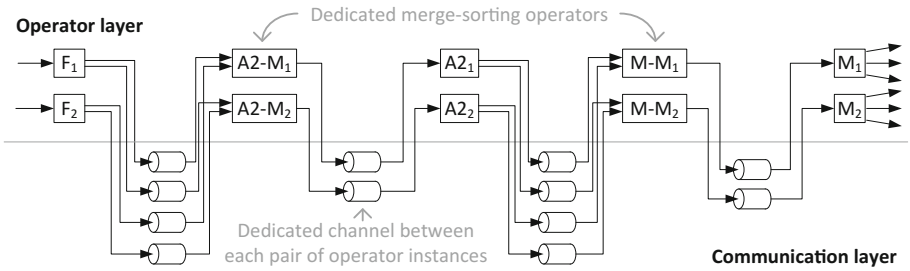


Fig. 2. Parallel query run by an SPE with operator-layer determinism.

3.1 Overheads of Operator-Layer Determinism

The deployment of dedicated merge-sorting operators in-between the query's operators results in an increase of the number of threads in SPEs such as Storm [26] or Flink [10] or in scheduling overheads for SPEs with schedulers ordering operators' execution [1,2,12], thus degrading throughput and increasing energy consumption. A lower throughput and a higher latency are also expected because of the increased number of operator instances and number of queues each tuple traverses. Using our example to provide an intuitive reasoning for the above claim, let us observe that each tuple traverses four queues and three operator instances from operator F to operator M (Fig. 2).

Moreover, merge-sorting operators might become the processing bottleneck. The maximum throughput of an operator instance can be observed as long as its preceding operators are not under-provisioned. That is, as long as the cost of its preceding merge-sorting operator is not a bottleneck. Unfortunately, the latter's cost (which is in the best case logarithmic in the number of input streams [15]) might be comparable to or higher than the query's operators. It should also be observed that, opting for a higher degree of parallelism when an operator cannot cope with its input rate might have a relapse on the throughput and latency of its downstream merge-sorting operator instances (which will have to merge-sort a higher number of input streams). For example in Fig. 2, suppose the processing cost of operator instances $A2_1, A2_2$ is higher than the cost of merge-sorting for operator instances $A2\text{-}M_1, A2\text{-}M_2$. The degree of parallelism for operator $A2$ could be increased to e.g. four instances. By doing this, each of the four instances of operator $A2\text{-}M$ would then be responsible for the merge-sorting of half of the input tuples. Nevertheless, each instance of the merge-sorting operator preceding operator M would now observe a higher cost for the merge-sorting of its input

tuples (coming from four rather than two input streams). Hence, increasing the degree of parallelism for $A2$ could overload the merge-sorting of tuples feed to M, thus decreasing, rather than increasing, the overall throughput of the query.

3.2 Benefits of Communication-Layer Determinism

The aim of communication-layer determinism is to avoid the deployment of merge-sorting operators in between each operator and its upstream peer instances. As shown in Fig. 3, this allows for the instances of operator F to be directly connected to those of operator $A2$. Since the merge-sorting would still need to be run to enforce determinism, a requirement of communication-layer determinism is to leverage threads that are already deployed by the SPE and share such operations rather than assigning them to a dedicated one, as this would in turn result in the previously discussed overheads. As discussed in [15], shared-memory merge-sorting can be carried out by multiple threads in a scalable fashion when the cost and the relapse that merge-sorting itself introduces is minimized by avoiding coarse-grained locking mechanisms.

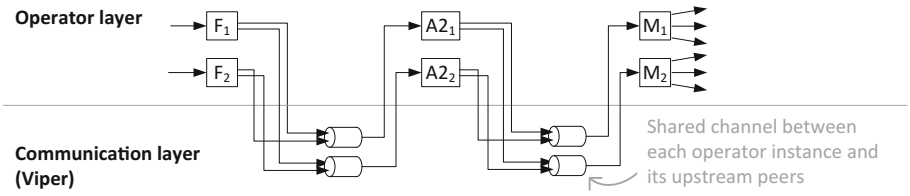


Fig. 3. Parallel query run by an SPE with communication-layer determinism.

3.3 The Viper Module

The Viper module allows for communication-layer determinism and provides an API defined by three main methods (Table 1). A channel is maintained at the Viper for any set of source operator instances $S_1, \ldots, S_m$ feeding a reader operator instance R (we use the term channel to refer to the data object used by a set of operator instances to share information, such an object can be a queue or another object). The channel, in our scheme, is either a thread-safe concurrent queue (when exactly one source S_1 and the reader R are connected) or a ScaleGate [15] object (when at least two source operators S_1, S_2 and the reader R are connected). Method `add` allows tuples from different sources to be merge-sorted into a single list, assuming that each source delivers tuples in non-decreasing timestamp order. Method `getReady` allows the list to be read in timestamp order by the reader guaranteeing that only *ready* tuples (cf. Definition 1) will be delivered. In this work, we extend the original ScaleGate proposing and integrating a flow-control approach using special watermark tuples [17] internally in the data structure. Such tuples are added periodically by the sources and

allow the readers to acknowledge the consumption rate to the sources, through a handshake mechanism, so that the latter can limit injection rate for slow readers.

With Viper, the merge-sorting cost is efficiently shared by the threads assigned to the instances of a parallel operator feeding the same downstream operator instance, thanks to its scalable probabilistically logarithmic lock-free implementation [15], which minimizes the necessary synchronization overheads [6].

Table 1. API of the Viper module

Method	Description
`void register(channel, sources, reader)`	Register a new *channel*, specifying its *sources* and the *reader* retrieving the timestamp-sorted stream of ready tuples
`void add(channel, sourceID, tuple)`	Add a *tuple* from a given *sourceID* to the specified *channel*
`tuple getReady(channel, readerID)`	Retrieve next ready *tuple* (if any) for the given *readerID* from the specified *channel*

4 Evaluation

To quantify the benefits of communication-layer determinism over operator-layer determinism, we integrated the Viper module in Apache Storm [26], studying its performance in terms of throughput (t/s), latency (ms) and energy consumption (mJ/t). In the following, we refer to operator-layer determinism as *OL* and communication-layer determinism as *CL*. We conducted our experiments on a dual-socket Intel Xeon E5-2687W 3.4 GHz server, with 8 cores per socket (yielding a total of 16 cores, 32 threads) and 64 GB of RAM. The server runs Scientific Linux 6.5 (5) based on the Red Hat Enterprise Linux operating systems. We used *likwid* [20] to read out RAPL Energy counters for the power metrics presented in our evaluation. All experiments have been run using Storm version 0.9.7 and OpenJDK Java version 1.8.0_91. The ScaleGate implementation is the one available at [23]. For channels accessed by a single source and reader (cf. Sect. 3), the Viper module relies on Java's `ConcurrentLinkedQueue`.

The evaluation runs the *Linear Road benchmark* [4], an established benchmark to study SPEs' performance that simulates vehicular traffic on a number of linear expressways, each composed of predefined *segments*. *Position reports* are forwarded every 30 s and carry the vehicle's *position* and *speed*. Vehicles are charged with a variable toll based on the traffic congestion level and the presence of *accidents*. The generated data is continuously processed to (i) detect possible accidents and (ii) compute tolls and notify vehicles. We provide the evaluation results for both a stateless (pos_rep) and a stateful (new_seg) operator of the benchmark. Operator pos_rep forwards an incoming tuple if it is a position report. Its selectivity is 0.99. Operator new_seg checks whether a vehicle is entering a new segment. Its selectivity is 0.34.

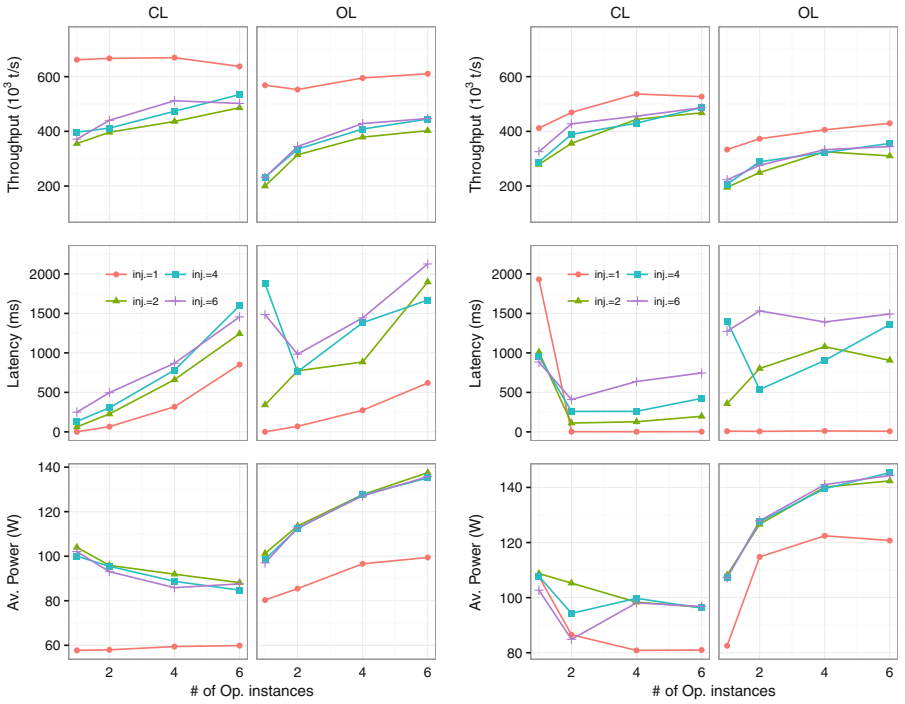
To study the performance of an operator, we start by deploying one instance of such operator together with one data injector and one sink. The injector is in charge of forwarding input tuples while maintaining the throughput statistics (per-second averages). The sink is in charge of maintaining latency statistics (per-second averages). This initial deployment allows us to measure the performance of the operator's centralized execution. The performance of its parallel counterpart depends on its parallelism degree (i.e., its number of parallel instances) and the parallelism degree of its upstream operator (i.e., the overhead introduced by deterministically merge-sorting the streams of the parallel upstream operator), as discussed in Sect. 3. For this reason, we increase the number of instances both for the injector and the operator to 2, 4 and 6 (i.e., we deploy 1 injector and 1, 2, 4 and 6 parallel operator instances, 2 parallel injectors and 1, 2, 4 and 6 parallel operator instances, ...) for a total of 16 configurations for each operator. The number of parallel sink instances deployed in each experiment is equal to the number of parallel instances of the operator in order for the former not to constitute a bottleneck. With OL-determinism provisioning, a merge-sorting operator is deployed for each instance of the operator if two or more injectors are deployed. Similarly, a merge-sorting operator is deployed before each instance of the sink if two or more parallel operator instances are deployed (no extra merge-sorting operators are needed for CL-determinism provisioning, using the Viper module). The highest degree of parallelism for the injector and operator is chosen so that the overall number of threads for both OL and CL that process and forward tuples is in the same order as the number of logical threads provided by the server.

For each configuration, we measure throughput as the number of tuples generated over each 5 s period and report the average throughput per second. The experiments are repeated 5 times; the reported values are averages over the runs of the same configuration.

4.1 Operator pos_rep

Figure 4a presents the performance results for the pos_rep operator for CL (left column) and OL (right column). Each sub-graph contains 4 lines, for 1, 2, 4 and 6 injectors, respectively. The upper sub-graphs present the throughput for the increasing number of instances of the parallel pos_rep operator. The middle sub-graphs present the latency while the lower ones present the energy consumption.

Given that operator pos_rep has a very high selectivity, almost each input tuple results in an output tuple. Since the operator is also a light stateless filtering operator, the cycles spent by it communicating (i.e., receiving and forwarding tuples) are higher than those spent processing tuples. Looking at the throughput performance of OL when one single injector is deployed, we can observe a stable throughput lower than 600,000 t/s. For the increasing number of operator instances, the latency increases to 600 ms (because the same output rate is shared by an increasing number of threads, thus resulting in longer times for output tuples to become *ready*) while the energy consumption stabilizes around 100 W. A similar behavior can be observed for a single injector



(a) Operator pos_rep. (b) Operator new_seg.

Fig. 4. Performance evaluation.

for CL, with a higher throughput that stabilizes at 650,000 t/s and a latency that also increases (to ∼800 ms). However, the energy consumption decreases to 60 W, 60% of that observed for OL, due of the channel shared by Viper between operator instances.

A different behavior can be observed for OL and CL when an increasing number of injectors is deployed. As shown in the figure, despite the lower throughput, due to the sorting overhead introduced to enforce determinism, CL results in a throughput growing over 500,000 t/s while OL stabilizes around 400,000 t/s. While still incurring in similar latency (lower in this case for CL than OL), OL's energy consumption grows up to 140 W while CL's achieves a consumption of 90 W due to the shared sorting work performed by the threads already deployed.

4.2 Operator new_seg

Using the same sub-graphs of Fig. 4a, b presents the performance results for OL and CL and operator new_seg. Differently for the stateless operator pos_rep, the stateful operator new_seg is characterized by a lower selectivity. This implies that, for the same input rate, the latter results in a lower output stream rate.

Given also its stateful nature, it results in an higher number of cycles spent processing rather receiving and forwarding tuples. As shown in Fig. 4b, the throughput achieved by CL is always higher than that of OL while observing a lower latency, both for the increasing number of injectors and the increasing number of operators. Also for this operator, CL achieves a throughput that is of approximately 100,000 t/s higher than that of OL. Finally, CL also results in lower power consumption, which does not exceed 100 W. On the other hand, OL's consumption grows to more than 140 W.

4.3 Power Consumption

Modern architectures deploy dynamic frequency scaling or CPU throttling where processors in idle state run at low frequency to conserve power and scale up the frequency on-demand. We observe in Fig. 4, that OL dissipates on average more power than CL. This is a result of differences in the number of threads utilized during a computation. With increasing number of execution threads, more cores are activated at high frequency which ultimately increases the power.

5 Related Work

Parallel execution of streaming operators has been first discussed by Flux [25] and implemented in StreamCloud [12,13]. The latter provided dedicated merge-sorting operators (added to queries by a dedicated compiler) to enforce deterministic execution at the operator layer, incurring the limitations discussed in Sect. 3. The techniques in [12,13,25] are now found in widely-adopted SPEs.

The communication-layer determinism we introduce in this paper is motivated by the increasing research interest in shared-memory parallelism. The most relevant advances, nonetheless, have so far been only tailored to Aggregates [14,24] and Joins [11,15,21,27]. The principles of the ScaleGate data object [23] have been proposed in [7] and leveraged in shared-memory parallelism for streaming aggregation [14] and joining [15]. In relation with our work, papers such as [3,22] discuss and provide evidence of the importance of careful design decisions for the internal communication mechanisms of SPEs. Differently from this work, nonetheless, optimizations focus on the reduction of unnecessary copies of tuples for the Borealis SPE in [3] (not considering determinism) and in a batching mechanism (complementary to the mechanism we propose) for Apache Storm.

6 Conclusions

Motivated by the observation that deterministic execution of streaming operators requires expensive synchronization to merge-sort streams from multiple operator instances (or data sources), we studied the limitations of operator-layer parallelism and how these can be overcome by communication-layer parallelism. Reducing the communication and synchronization costs among operator instances running within an SPE is a key factor in boosting its scale up potential.

In this paper, we propose a module, which we call Viper, that encapsulates and reduces the aforementioned costs, enabling for deterministic execution to be provided in a transparent way by the communication layer of an SPE. We provide evidence that such a module can be leveraged by SPEs, by integrating it into Apache Storm, which is a representative SPE of one-at-a-time analysis paradigm, for ultra-low latency processing. Our evaluation shows that the throughput of parallel operators interconnected with the Viper module increases by up to 70% and results in half of the energy consumption.

Acknowledgments. This work was supported by the Swedish Foundation for Strategic Research under the project "Future factories in the cloud (FiC)", grant number GMT14-0032 and the Swedish Research Council (Vetenskapsrådet) projects "HARE: Self-deploying and Adaptive Data Streaming Analytics in Fog Architectures" Contract nr. 2016-03800 and "Models and Techniques for Energy-Efficient Concurrent Data Access Designs" Contract nr. 2016-05360.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., et al.: The design of the borealis stream processing engine. In: CIDR, vol. 5, pp. 277–289 (2005)
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.: Aurora: a new model and architecture for data stream management. VLDB J. Int. J. Very Large Data Bases **12**(2), 120–139 (2003)
3. Akram, S., Marazakis, M., Bilas, A.: Understanding and improving the cost of scaling distributed event processing. In: Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems, pp. 290–301. ACM (2012)
4. Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A.S., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear road: a stream data management benchmark. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, pp. 480–491. VLDB Endowment (2004)
5. Balazinska, M., Balakrishnan, H., Madden, S.R., Stonebraker, M.: Fault-tolerance in the Borealis distributed stream processing system. In: ACM TODS (2008)
6. Cederman, D., Chatterjee, B., Nguyen, N., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: A study of the behavior of synchronization methods in commonly used languages and systems. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS), pp. 1309–1320. IEEE (2013)
7. Cederman, D., Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: Brief announcement: concurrent data structures for efficient streaming aggregation. In: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2014, pp. 76–78. ACM (2014)
8. David, H., Gorbatov, E., Hanebutte, U.R., Khanna, R., Le, C.: RAPL: memory power estimation and capping. In: Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design, ISLPED 2010, pp. 189–194. ACM, New York (2010)
9. De Matteis, T., Mencagli, G.: Keep calm and react with foresight: strategies for low-latency and energy-efficient elastic data stream processing. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, pp. 13:1–13:12. ACM, New York (2016)

10. Apache Flink. https://flink.apache.org/
11. Gedik, B., Bordawekar, R.R., Philip, S.Y.: CellJoin: a parallel stream join operator for the cell processor. VLDB J. **18**(2), 501–519 (2009)
12. Gulisano, V.: StreamCloud: an elastic parallel-distributed stream processing engine. Ph.D. thesis, Universidad Politécnica de Madrid (2012)
13. Gulisano, V., Jimenez-Peris, R., Patino-Martinez, M., Valduriez, P.: StreamCloud: a large scale data streaming system. In: 2010 IEEE 30th International Conference on Distributed Computing Systems (ICDCS), pp. 126–137. IEEE (2010)
14. Gulisano, V., Nikolakopoulos, Y., Cederman, D., Papatriantafilou, M., Tsigas, P.: Efficient data streaming multiway aggregation through concurrent algorithmic designs and new abstract data types. CoRR, abs/1606.04746 (2016)
15. Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., Tsigas, P.: ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join. IEEE Trans. Big Data (99) (2016)
16. Gulisano, V., Nikolakopoulos, Y., Walulya, I., Papatriantafilou, M., Tsigas, P.: Deterministic real-time analytics of geospatial data streams through ScaleGate objects. In: Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems, DEBS 2015, pp. 316–317. ACM, New York (2015)
17. Johnson, T., Muthukrishnan, S., Shkapenyuk, V., Spatscheck, O.: A heartbeat mechanism and its application in gigascope. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 1079–1088. VLDB Endowment (2005)
18. Kalyvianaki, E., Fiscato, M., Salonidis, T., Pietzuch, P.: THEMIS: fairness in federated stream processing under overload. In: Proceedings of the 2016 International Conference on Management of Data, pp. 541–553. ACM (2016)
19. Koliousis, A., Weidlich, M., Castro Fernandez, R., Wolf, A.L., Costa, P., Pietzuch, P.: SABER: window-based hybrid stream processing for heterogeneous architectures. In: Proceedings of the 2016 International Conference on Management of Data, pp. 555–569. ACM (2016)
20. LIKWID: Performance measurement and benchmark suite. https://github.com/RRZE-HPC/likwid
21. Roy, P., Teubner, J., Gemulla, R.: Low-latency handshake join. Proc. VLDB Endow. **7**(9), 709–720 (2014)
22. Sax, M.J., Castellanos, M., Chen, Q., Hsu, M.: Aeolus: an optimizer for distributed intra-node-parallel streaming systems. In: 2013 IEEE 29th International Conference on Data Engineering (ICDE), pp. 1280–1283. IEEE (2013)
23. ScaleGate. https://github.com/dcs-chalmers/scalegate
24. Schneidert, S., Andrade, H., Gedik, B., Wu, K.-L., Nikolopoulos, D.S.: Evaluation of streaming aggregation on parallel hardware architectures. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems, pp. 248–257. ACM (2010)
25. Shah, M.A., Hellerstein, J.M., Chandrasekaran, S., Franklin, M.J.: Flux: an adaptive partitioning operator for continuous query systems. In: Proceedings of the 19th International Conference on Data Engineering, pp. 25–36. IEEE (2003)
26. Apache Storm. http://storm.apache.org/
27. Teubner, J., Mueller, R.: How soccer players would do stream joins. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (2011)