Accelerating Stream Processing Queries with Congestion-aware Scheduling and Real-time Linux Threads

Citation for the original published paper (version of record):

Frasca, F., Gulisano, V., Mencagli, G. et al (2023). Accelerating Stream Processing Queries with Congestion-aware Scheduling and Real-time Linux
Threads. Proceedings of the 20th ACM International Conference on Computing Frontiers 2023, CF 2023: 144-153. http://dx.doi.org/10.1145/3587135.3592202

Accelerating Stream Processing Queries with Congestion-aware Scheduling and Real-time Linux Threads

Fausto Frasca
University of Pisa
Pisa, Italy
f.frasca4@studenti.unipi.it

Vincenzo Gulisano
Chalmers University of Technology
Göteborg, Sweden
vincenzo.gulisano@chalmers.se

Gabriele Mencagli
University of Pisa
Pisa, Italy
gabriele.mencagli@unipi.it

Dimitris Palyvos-Giannas
Chalmers University of Technology
Göteborg, Sweden
palyvos@chalmers.se

Massimo Torquati
University of Pisa
Pisa, Italy
massimo.torquati@unipi.it

ABSTRACT

Stream Processing Engines (SPEs) have been used by companies and industries to develop queries able to extract insights from data streams. The Edge/IoT context poses additional challenges, since streaming queries need to run closer to data producers to save latency, i.e., on resource-constrained devices. Lachesis is a middleware helping Linux to schedule more efficiently threads of the SPE, which revealed useful especially for devices with limited CPU resources. Lachesis does not require any architectural change to the SPE implementation. It collects metrics from the SPE, and computes high-level priorities that are converted into hints to the Operating System to affect its actual scheduling of threads. This paper extends the initial contribution of Lachesis in two main directions: *i)* we optimize the policy assigning to threads a priority proportional to their actual load by accurately studying the implementation of Storm and Flink, two popular SPEs; *ii)* instead of restricting the OS scheduling to traditional `SCHED_OTHER` threads as done previously by Lachesis, we leverage the real-time capability of the modern Linux kernel. Our experimental evaluation shows that both enhancements provide important benefits compared with the previous version of Lachesis: we get +9.75% (average) throughput (+19% peak) with −27% latency on average (−40% peak).

CCS CONCEPTS

• **Information systems → Online analytical processing engines**; • **Software and its engineering → Scheduling**.

KEYWORDS

Data Stream Processing, Apache Storm, Apache Flink, Real-time Threads, Linux Scheduler

ACM Reference Format:
Fausto Frasca, Vincenzo Gulisano, Gabriele Mencagli, Dimitris Palyvos-Giannas, and Massimo Torquati. 2023. Accelerating Stream Processing Queries with Congestion-aware Scheduling and Real-time Linux Threads. In *20th ACM International Conference on Computing Frontiers (CF '23), May 9–11, 2023, Bologna, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3587135.3592202

1 INTRODUCTION

Data has become one of the world's most valuable commodities, with many business activities taking advantage of data to extract insights and hidden value driving decision-making processes. Moreover, with the proliferation of sensing devices, high-volume data is often available in the form of *streams*, i.e., unbounded sequences of data items produced with high speed. However, efficiently coping with unbounded streams instead of static datasets poses challenges both from the algorithmic perspective and regarding the implementation of data processing pipelines, which should be capable of dealing with streams by exploiting parallel hardware.

To enable Data Stream Processing (DSP), first-generation *Stream Processing Engines* (SPEs) like Aurora [3] extended DBMSs to support streams of structured records, which were modeled as particular kinds of unbounded relations queried with extensions of the SQL query language and transformed by relational operators (e.g., selection, aggregates, joins). The diffusion of parallel and distributed architectures such as clusters and Clouds, and the need to analyze unstructured streams with arbitrary non-relational operators, have paved the way to more general scale-out SPEs such as Apache Storm [2] and Flink [1].

When stream sources are geographically distributed, transferring all streams to a Cloud requires high network provisions (in terms of bandwidth and reliability), which makes low latency a challenge in such contexts. To overcome this issue, the idea is to execute stream analysis in resources closer to data sources, acknowledging though that they are less computationally powerful than Clouds (e.g., IoT devices with a small number of CPU cores, small memory capacity, and energy constraints) and require to adapt or rethink the SPEs originally designed for traditional servers or Clouds.

A relatively small amount of papers have tried to improve the existing SPEs in order to better leverage edge resources. A work of this kind, which inspired our research, is EdgeWise [11], a patched version of Storm for edge resources. As stated by the authors, most

CF '23, May 9–11, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0140-5/23/05…$15.00
https://doi.org/10.1145/3587135.3592202

of the existing SPEs adopt the so-called *One Worker Per Operator Architecture* (OWPOA), where each operator of the application is run by a dedicated thread, and different operators mapped onto distinct threads communicate by means of shared queues. Since most of the streaming queries have many operators (and so many threads are spawned by the SPE runtime system), the effective scheduling of those threads onto the available CPU cores (especially when cores are relatively scarce as in Edge/IoT platforms) plays a relevant role in terms of overall performance. Nonetheless, existing SPEs simply rely on the Operating System (OS) scheduler to choose which operator to run next. EdgeWise changes this paradigm by setting up a custom version of Storm where its OWPOA design has been replaced with a user-space scheduler that dynamically assigns operators (considered as logical tasks) to a fixed-size pool of threads, so bypassing OS scheduling. This approach is intrusive, because it requires deep changes to the official implementation of Storm, and it is not portable to newer versions of this SPE. Lachesis [13] is a new system that requires no modification to the SPE (and to its OWPOA architecture) but instead steers the OS scheduling by using the *nice* command to properly prioritize threads running streaming queries (i.e., giving hints to the OS in order to customize its scheduling instead or bypassing it with a user-space scheduler).

This paper extends this work by proposing a new version of Lachesis having the following novel contributions:

- the congestion-aware policy assigns to each thread running a query operator a priority related to its actual load, estimated as the length of the queue of inputs to be read and processed by the logical component of the SPE running that operator. By conducting a more in-depth analysis of the internal implementation of Storm and Flink, two popular open-source SPEs, we identify different threads responsible to run several computing and communication steps of stream processing applications, and we find more precise approaches to schedule them according to their actual load;
- we study how to enforce the Linux real-time scheduling features to promote the SPE threads running streaming applications as real-time threads, and deal with the assignment of their priorities to bring better performance.

The experimental evaluation will analyze these optimizations of Lachesis using applications belonging to different benchmark suites (the LinearRoad benchmark [6] and the RiotBench [16]), running on Odroid boards. Furthermore, we consider two different SPEs, Storm and Flink, so confirming the effectiveness of Lachesis owing to the generality of its design principles, which can be applied to all systems respecting the OWPOA design, which is currently adopted by all major open-source SPEs.

In the next part, we introduce the background of this paper. Then, we review what Lachesis is and its architecture. Next, we describe our enhancements in § 4 and we provide the experimental evaluation in § 5. Finally, we provide a description of related works in § 6, and we draw the conclusion of the paper in § 7.

2 BACKGROUND

In this section, we review the basic concepts related to DSP, and the runtime system of two popular SPEs: Storm and Flink. Furthermore, we provide a brief overview of the Linux scheduler.

2.1 Stream Processing Engines

DSP applications are data-flow graphs whose vertices represent *operators* and *edges* model streams [5]. Operators perform intermediate transformation stages working on inputs from other operators and are able to produce outputs directed to other vertices of the graph. Streams convey inputs represented as *tuples* of attributes in a structured manner, although unstructured data can also be represented and computed by modern streaming systems.

DSP popularity has improved with the advent of scale-out distributed SPEs providing a user-friendly manner to develop streaming applications and to transparently run them on a cluster of distributed machines or in the Cloud. As hinted in the Introduction, the standard architecture of such modern SPEs follows the so-called OWPOA-based paradigm, which is depicted in Fig. 1. While streaming applications are dataflows of operators, the SPE runtime system runs each operator by a dedicated thread. Threads, connected in a pipeline manner through shared queues implementing data exchange between corresponding operators, are eventually run by the OS scheduler on top of the available cores of the machine.

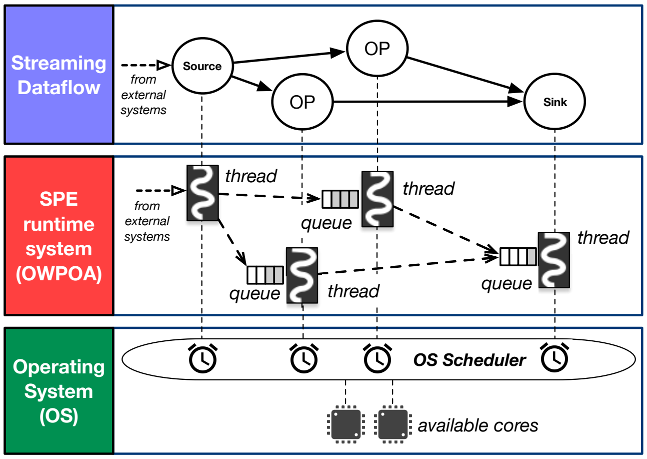

Figure 1: Streaming dataflows, generic OWPOA-based SPE, and OS scheduler.

Some deviations might be possible w.r.t the previous general picture. Notably, more threads can be used for each operator: for example, one thread is devoted to reading new inputs from the input queue and computing the user-specified transformation logic associated with the operator, while a so-called *helper thread* is in charge of transmitting the produced output results to the input queue of the right destination operator. Furthermore, the dataflow can be partitioned among different machines. In that case, the SPE runtime system consists of more processes (within and across machines), each spawning threads for the operators of the local partition while communications crossing the process boundaries are implemented through TCP/IP connections across the network.

We provide next an in-depth view of the runtime system of two popular SPEs, Storm and Flink, showing the mapping between the abstract concepts in Fig. 1 with the specific terminology adopted by those SPEs. However, we note that Lachesis's contribution also applies to other OWPOA-based SPEs, as shown in [13].

2.1.1 *Apache Storm.*

Storm is a distributed SPE. A Storm cluster is composed of a primary node named Nimbus and secondary nodes, each running a Supervisor daemon process. An application, in the form of a `jar` compiled file, is distributed by the primary node across the secondary nodes by spawning proper *Worker* JVM processes on each machine. Each Worker executes a subset of the operators of the application, by running separate *Executor* threads. The programmer can instrument the code to decide how many Executors to run per operator, so replicating them to increase throughput. A more in-depth view of the Storm runtime system is sketched in Figure 2.

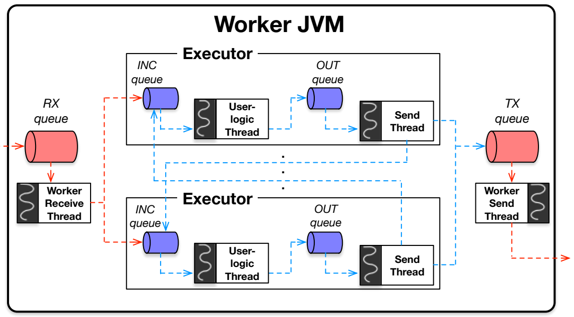

Figure 2: In-depth view of the Storm runtime system.

Each Worker has a *RX queue* containing serialized tuples coming from other Workers. A *Worker receive thread* is in charge of extracting inputs from this queue, deserializing and copying them into the *INC queue* of the right destination Executor. According to the abstract SPE picture in Fig. 1, each Executor is composed of two threads: the first is in charge of running the user-defined logic of the corresponding operator, to transform inputs into outputs pushed to its *OUT queue*; the second, called *send thread*, is a helper thread that pops results from the *OUT queue* and copies them either into the *INC queue* of another Executor or to the *TX queue* of the Worker process, if results must be delivered (by the *Worker send thread*) to operators run by other JVMs.

2.1.2 *Apache Flink.*

Flink architecture has a primary node (JobManager) and secondary nodes (TaskManagers). Flink is able to chain a sequence of operators connected in a pipeline. Each chained sequence of operators is called a *Subtask* (which can contain also one operator only if chaining cannot be applied), and each Subtask is executed by a dedicated thread within the TaskManager JVM process. Figure 3 shows the TaskManager architecture.

Data transferring between Subtasks of different TaskManagers is performed using dedicated TCP connections managed by Netty. Each Subtask has a *main thread* that performs the data deserialization from input buffers, the computation of its chained operators, and pushes results (after serializing them) into proper output buffers. As soon as an output buffer is full, it is given to the *Netty client* whose threads (not part of Flink) consume results in the buffer and send them to other TaskManagers. To avoid notifying Netty only when buffers are full, an *output flusher thread* (a sort of helper thread according to our previous abstract terminology) is in charge of notifying Netty as soon as a timer expires to reduce latency.

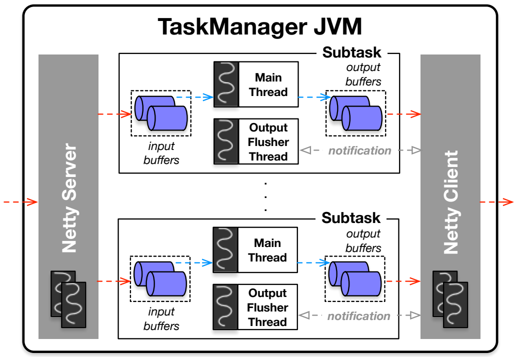

Figure 3: In-depth view of the Flink runtime system.

2.2 Linux Scheduler

Linux is a popular OS. The Linux scheduler is the kernel component that selects which process or thread to run next on the available cores of the machine. As process and thread are not differentiated inside the kernel for scheduling purposes, we refer in the following to *tasks* as schedulable entities. The scheduler distinguishes between different classes of tasks. `SCHED_NORMAL` is the class of normal tasks whose scheduling is implemented by the *Completely Fair Scheduler* (CFS) component. The classes `SCHED_RR` and `SCHED_FIFO` are the ones handled by the *Real-time Scheduler* (RTS).

2.2.1 *CFS.*

This scheduler approximates perfect multitasking, i.e., if there are N tasks runnable, they should receive $1/N$ of the total compute power each. For each task, CFS maintains a set of information. One meta-data field is named `vruntime`, and represents the total amount of time the task has run so far weighted by a proper `load` factor. CFS implements the *run queue* of schedulable tasks as a red-black self-balancing tree indexed by the `vruntime` parameter, in such a way that tree nodes (representing tasks) having the smallest `vruntime` are placed in the left-most part. As a task advances its execution, its `vruntime` parameter is updated, and every time the scheduler needs to pick up the next schedulable task, it extracts the one having the smallest `vruntime`. Linux users can influence CFS decisions through the `nice` command. This command accepts an integer between -20 and 19 (the higher the number, the lower the priority), and changes the `load` parameter used to compute the `vruntime`. Increasing/decreasing the `nice` level of one unit yields about a ±10% variation in the `vruntime` value of the task [12].

2.2.2 *RTS.*

Real-time tasks are always prioritized over normal tasks. This means that as soon as there is a runnable real-time task to execute, it will be given access to the processor even if normal tasks are waiting to run for a long time. Each real-time task is assigned to a *static priority* between 1 and 99 (the higher the number, the higher the priority), and the kernel keeps a separate list for each of these priorities. The priority is not changed by the kernel dynamically, but it is under user control and can be modified at runtime via the `chrt` command. The kernel differentiates tasks belonging to the `SCHED_RR` and `SCHED_FIFO` classes. In both cases, a running task can be preempted by another runnable task having a higher priority. However, `SCHED_RR` tasks are executed for a maximum time-slice in a circular manner among other tasks having the same priority. The time-slice can be controlled with the `sched_rr_timeslice_ms` parameter located at `/proc/sys/kernel/` (default is 25 ms). Instead, the `SCHED_FIFO` class is represented by real-time tasks that are executed until they explicitly yield the processor or they are preempted by a task with higher priority.

3 LACHESIS ARCHITECTURE

We want to help Linux to schedule threads of streaming queries in a more effective manner. Instead of balancing the CPU time as done by the Linux CFS, our goal is to continuously acquire measurements from the running queries to tune the kernel parameters that affect how Linux performs their scheduling. This has been investigated in the previous version of Lachesis [13] by relying on the `nice` command. The value of the `nice` parameter for each thread within the SPE is periodically updated on the basis of measurements exposed by the SPE monitoring API, which represent the actual load of the threads that run streaming queries in the SPE. Lachesis has a component-based architecture depicted in Figure 4.

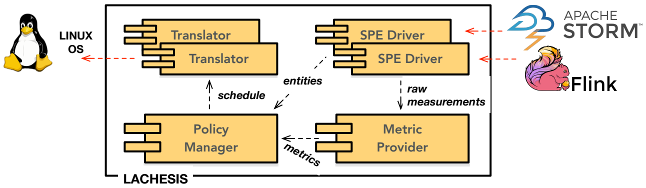

Figure 4: Architecture and components of Lachesis.

SPE Drivers. Lachesis incorporates a set of drivers, one for each supported OWPOA-based SPE, acting as a bridge between Lachesis and the SPE. They provide two types of information to Lachesis: *i)* identifiers of the entities of the SPE runtime system that are of interest to Lachesis (Worker and TaskManager PIDs, identifiers of threads within Executors in Storm or Subtasks in Flink, see § 2.1); *ii)* raw metrics from the SPE (e.g., number of input tuples currently present in the INC queues of the Executors in Storm, number of outputs produced per input, and so forth). The driver retrieves such metrics, without requiring any change to the SPE implementation, through the public metric API exposed by the specific SPE directly or through external tools like Graphite.

Metric Provider. It receives raw measurements from the drivers. It converts raw measurements into (aggregated) metrics that can be used by Lachesis to drive the scheduling. For example, if the scheduling policy adopted by Lachesis needs to be aware of the *selectivity* of operators in the query graph (i.e., the average amount of outputs produced per input by the operator), this metric might be not directly available from the driver (and from the metric API of the SPE), but can be derived by knowing the number of inputs consumed and outputs produced per operator at each time instant.

Policy Manager. It computes, at execution intervals called *scheduling rounds*, the high-level priorities for all threads running streaming queries within the controlled SPEs. We call a *schedule* an assignment of high-level priorities to threads. The built-in policy of Lachesis is the *congestion-aware policy* adopted (and theoretically demonstrated) in EdgeWise [11] for its user-level scheduler, and now used to help the OS. The goal of the policy is to prioritize threads whose operators have more input tuples in their queues, so balancing the size of the queues to achieve higher throughput and lower latency. We show that this policy reveals effective without introducing a user-space scheduler, but steering the OS scheduler to reflect the policy behavior by controlling the way in which Linux performs the assignment of priorities to threads.

Translators. They convert high-level priorities into commands that tune the Linux scheduler. The translator used by Lachesis is based on the `nice` command. Since `nice` values are integers in a limited range (see § 2.2.1), this translator performs a min-max normalization to convert high-level priorities into admissible `nice` values. When used with *control groups* (`cgroups`), a Linux feature used by containers, the *nice* prioritization assigned to threads in the same group respects the CPU limit of the group (e.g., controlled via the `cpu.cfs_quota_us` parameter). So Lachesis, together with streaming queries and the SPE, can be deployed into containers, which is a common scenario in the Edge/IoT context.

4 LACHESIS OPTIMIZATIONS

This paper presents and evaluates different enhancements to improve Lachesis[1]. We extend how Lachesis implements the congestion-aware scheduling policy by looking more deeply at the internal structure of the SPE runtime system, and we extend the set of translators to go beyond the use of `SCHED_NORMAL` threads controlled by `nice`, but focusing on the real-time capabilities of Linux.

4.1 Refining the Congestion-aware Policy

The policy assigns higher priorities to operators in the query that represent bottlenecks, i.e., that have a longer queue of inputs to consume. The general goal is to balance the length of the queues and push the sustainable point of the SPE to a higher input rate. This idea has been originally implemented in EdgeWise [11]. EdgeWise changes the Storm runtime system intrusively by introducing a user-level scheduler, so transforming operators into user-level tasks whose scheduling on a fixed-size pool of kernel-level threads is controlled by the SPE itself (and not by the OS). As demonstrated in [13], Lachesis outperforms EdgeWise by applying the same policy without introducing a user-level scheduler, so without changing the official OWPOA implementation of Storm. The policy is implemented as follows (steps executed at each scheduling round):

- Lachesis gets the updated queue lengths of the operators in the running query;
- for Storm, all the threads within each Executor are associated with the current length of the corresponding INC queue (see Figure 2) received from the Metric Provider. Analogously for Flink, all threads of each Subtask are associated with the number of tuples currently in the input buffers (see Figure 3);
- these queue lengths represent the high-level priority values of the policy, and so the schedule is given to a translator, which applies a min-max normalization and converts the priorities into `nice` values. The list of `nice` commands is executed by Lachesis while the query is running.

[1] This version of Lachesis is available at https://github.com/ParaGroup/Lachesis-RT

To improve this strategy, we look at two important optimizations described in the following.

4.1.1 Distinguishing threads within the SPE. In Storm an Executor is composed of two threads, the one running the user logic on each input, and the second responsible to read the OUT queue and delivering results either to the TX queue of the Worker process or to the INC queue of another Executor of the same Worker (see Figure 2). The INC and the OUT queues of the same Executor might have completely different lengths. Think about an operator with high selectivity, i.e., capable of producing many outputs per input. In that case, even if the INC queue length might be small, the OUT queue length might be significantly large. The consequence is that the *user-logic thread* and the *send thread* of the same Executor should be prioritized in a different manner, a possibility that is not supported by the Lachesis version presented in [13], where all threads of the same Executor received a priority proportional to the length of the INC queue only. The same concept applies to Flink with the *main thread* and the *output flusher thread* of Subtasks.

To change the policy, several components of Lachesis need to be modified. The first modification is related to the way Lachesis uses to retrieve the metrics from the SPE. Essentially, metrics are collected by the SPE Driver through Graphite, which is used as a hub for all metrics generated by the SPE. With a default interval of one second, Lachesis sends a request to Graphite to get the last 10-second values of a specific metric (e.g., the length of a specific queue). Once the answer has been collected, Lachesis computes the average value, which is used to apply the policy. We modify the `GraphiteDataFetcher` class to ask Graphite for a list of metrics with a single multi-query request. To provide those metric values to the Policy Manager, we modified the meta-data used to identify the metrics in order to discern whether they refer to INC or OUT queues of Executors (Storm) or Subtasks (Flink).

A further effort was made to make more homogeneous the way in which queue lengths are measured in Storm. In the default configuration, the capacity of INC and OUT queues are fixed to the same value of 1024 entries. However, while the elements of OUT queues are single tuples, the ones of INC queues are lists (batches) of several contiguous tuples (by default, 100). Therefore, Lachesis has been modified to standardize the capacity of these queues before running any query. This has been done by properly setting the `topology.executor.send.buffer.size` configuration parameter of the OUT queue capacity to reflect the same capacity of INC queues in terms of individual tuples.

4.1.2 Dealing with source operators. Another limitation relates to how source operators are considered. These operators do not receive inputs from other operators, but rather from external systems. In Storm, the length of the INC queue of Executors running source operators will always return zero, although such sources can be highly loaded. Indeed, their actual input queue is invisible to Storm, and so not accessible with the available built-in metrics. Therefore, source operators might receive a priority generally lower than the rest of the operators in the graph, which does not reflect their load.

To solve this issue, we extend Lachesis to collect metrics received from a new set of source operators that we implemented in Storm and Flink. They report to Graphite, as custom metrics, the actual size of the queue(s) that such operators actually use in their code to buffer incoming data from external systems before delivering them to the rest of the DAG. We consider two kinds of operators although the approach can be extended to others too:

- `FileSource`: it extends the unbounded file source of Flink (analogously the `FileReadSpout` in Storm). It reads raw input data from a given file (or set of files), parses the entries, and delivers tuples to the DAG. This source uses a bounded queue to store raw entries that need to be parsed and delivered, and periodically report such a metric to Graphite;
- `KafkaSource`: since streaming queries usually receive data from publish-subscribe systems like Apache Kafka, we extend the `KafkaSource` and `KafkaSpout` that allow Flink and Storm to subscribe to Kafka topics respectively. Again, these sources report to Graphite the actual size of their internal queues as a new custom metric.

4.2 Scheduling Real-time Threads

The second optimization is to use the real-time scheduler of Linux. As explained in § 2.2.2, `SCHED_RR` and `SCHED_FIFO` tasks always receive a higher priority than `SCHED_NORMAL` tasks, and can be controlled through a set of system calls to manipulate the real-time attributes (through the `chrt` shell command). To exploit such a feature, we add a new translator that converts the high-level priorities computed by the Policy Manager into a list of real-time priorities (each an integer between 1 and 99), which are then applied to the underlying OS. The translator uses `SCHED_RR` tasks, which allow fairer scheduling among tasks with the same priority.

Reliable use of this translator is made possible by the *real-time group scheduling* feature of Linux kernels, since otherwise real-time threads could starve the machine and were often limited to privileged processes. With such a feature, both the SPE with the running queries and Lachesis can be placed in a container, which can receive a specific amount of real-time CPU time through the `cpu.rt_runtime_us` parameter of `cgroups`. Doing so, threads of the query (even if they are promoted as real-time threads) cannot spend more time than this limit per one period of time. Therefore, this new translator represents an alternative still usable way Lachesis can adopt to control thread scheduling in Linux.

5 EXPERIMENTAL EVALUATION

This section is devoted to assessing the effectiveness of the refinement of the congestion-aware policy and the use of the new real-time translator. In order to evaluate each optimization on its own, and then highlight the joint contribution of both, we evaluate in § 5.2 the impact of the refinement of the congestion-aware policy with standard CFS scheduling of `SCHED_OTHER` threads. Then, in § 5.3, we present the further improvement gained by exploiting the real-time scheduling features of the Linux kernel.

5.1 Experimental Setup

Queries. We consider a set of queries presented in the literature and available in independent repositories. The first (`LR`) is taken from the LinearRoad benchmark [6]. It is a tolling system for motor vehicle expressways with nine operators. The query is provided for both Storm and Flink. The query has a realistic setup where input streams are generated by reading from a Kafka broker. We also use two queries from the RIoTBench suite [16]. ETL is an extract-load-transform pipeline of ten operators that processes IoT sensor streams. STATS consists of ten operators that apply different kinds of statistical analytics to the input stream. For both ETL and STATS we use the official code-base in GitHub[2], where the two applications are available in Storm and raw tuples are read from dataset files.

Hardware. Since Lachesis aims at enhancing streaming queries on low-end devices, we use an Odroid-XU4 board with four ARM Cortex-A15 2GHz and four Cortex-A7 1.4GHz. Storm and Flink are executed on the four high-performance cores, while Graphite and Lachesis have been pinned on the four high-efficiency cores. For the LR query, the Kafka broker is executed on a machine in the same LAN. We have replicated our experimental results on different versions of the Linux kernel (i.e., 5.4.167-240 and 4.14.180-178). For the sake of space, this section will discuss the results obtained on the more recent kernel version.

Baselines. We evaluate the new implementation of the congestion-aware policy with the nice translator (Lachesis-Mod) and with the new real-time translator (Lachesis-Mod-RT). Both are compared against the prior version of Lachesis [13], so without the policy optimizations and based on the nice translator, which we refer to as Lachesis. We report also the results with EdgeWise, which is available in Storm for the ETL and STATS queries only [11].

Performance Metrics. We focus on *throughput*, i.e., the average number of input tuples that the SPE processes per second. We also consider two definitions of latency. The first one is the *end-to-end* (e2e) *latency*, which represents the full elapsed time from when an input item has been produced (e.g., by an external software such as a Publish-Subscribe system) to when the corresponding final output has been received by a Sink operator. Since it incorporates all components crossed from inputs to outputs, e2e latency is the meaningful metric from the user's perspective. However, for the sake of understanding Lachesis behavior, we also consider the intra-SPE latency (simply *latency*), which excludes the elapsed time between the external system feeding the SPE and the SPE itself.

5.2 Lachesis-Mod with SCHED_NORMAL Threads

We first compare Lachesis-Mod against the old Lachesis version in [13]. Furthermore, we compare against the OS and EdgeWise. All plots in this paper show the effect of higher input rates (i.e., the speed at which inputs are generated) on throughput and latency and/or e2e latency. Both latencies are shown using a logarithmic scale for the y-axis, in order to highlight their general trend over the whole space of input rates used. The expected Lachesis-Mod contribution is to achieve higher throughput and to sustain higher input rates with lower (at least end-to-end) latency.

5.2.1 Storm results. Figure 5 shows the results with the ETL query by reporting, as a function of the input rate, the throughput, latency, e2e latency, and the policy goal. The latter is the coefficient of variation of the length of all the queues in the SPE runtime system (the lower, the better since queue lengths are more homogeneous).

Lachesis-Mod keeps the system stable up to a rate of 1,875 tuple/sec, and then degrades more smoothly than Lachesis. At the new saturation point, throughput is +4.5% higher than Lachesis and +39% than OS, which is unstable with a rate greater than 1,375 tuple/sec. In terms of latency and e2e latency, with a rate of 1,875 tuple/sec Lachesis-Mod exhibits −95% lower latency (below the second) compared with Lachesis, since it maintains the lengths of the queues more homogeneous as reflected by the Policy Goal plot. The e2e latency looks similar, with a decrease of −98% at the new saturation point. EdgeWise provides better throughput than OS. We experienced that with input rates higher than 2,000 tuple/sec EdgeWise crashes. However, with lower rates we observe that throughput and latency are much better with Lachesis and Lachesis-Mod, confirming that the intrusive changes to introduce a user-space scheduler by EdgeWise do not pay off.

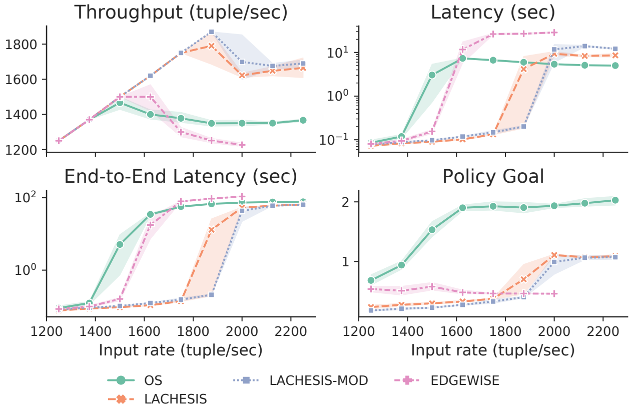

Figure 5: Results with the ETL query on Storm.

The results of the STATS query are shown in Figure 6. OS reaches the saturation point around 320 tuple/sec and its throughput stops increasing at a rate higher than 330 tuple/sec. Lachesis improves throughput up to a rate of 350 tuple/sec and beyond that limit it degrades smoothly. Lachesis-Mod provides an increasing throughput up to a rate of 400 tuple/sec, so showing a lower saturation. With the maximum considered rate Lachesis-Mod achieves +22% higher throughput, −9% lower e2e latency than Lachesis, while OS performs poorly. With high input rates both throughput and latency degrade very quickly with EdgeWise.

The LR query shows a different scenario, since the raw input stream is read from Kafka. As shown in Figure 7, the throughput improvement achieved by Lachesis-Mod is here marginal (+0.35% better than Lachesis with 7,500 tuple/sec) while the advantage w.r.t the OS is still remarkable.

Since Figure 7 shows latencies in logarithmic scale, we provide a more detailed view around the saturation point in Figure 8 . We notice that Lachesis-Mod provides lower e2e latencies compared to Lachesis while we experience the opposite behavior for the latency. According to our latency definitions, we can say that $e2e\_latency = wait_{external} + latency$, so the sum between the average waiting time between the data producer and the SPE and the source-to-sink internal latency spent within the SPE. In the case of Lachesis-Mod, threads of the source receive a number of CPU cycles proportional to their actual load (which in turn depends on the actual size of the external queues/buffers), while in Lachesis the nice values

[2] RiotBench benchmark suite is available at https://github.com/dream-lab/riot-bench

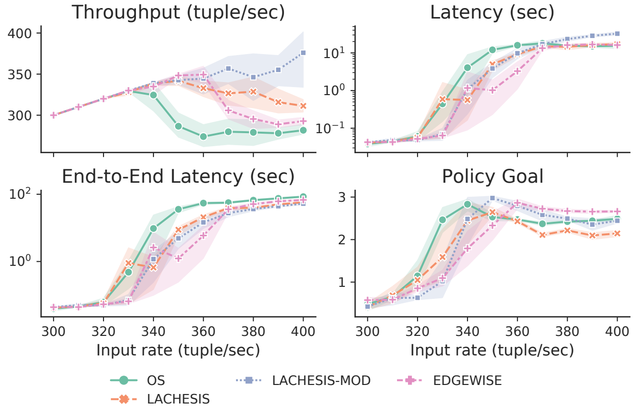

Figure 6: Results with the STATS query on Storm.

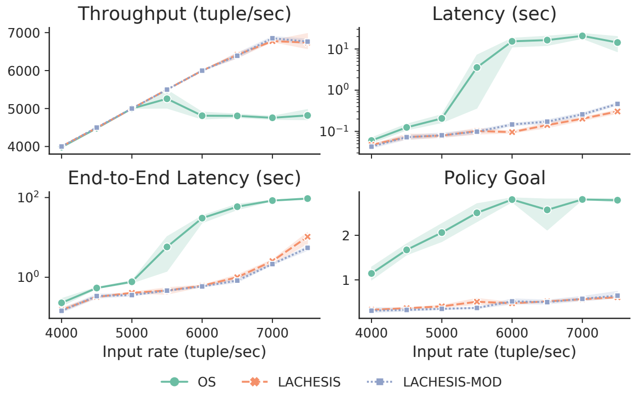

Figure 7: Results with the LR query on Storm.

of those threads were assigned by assuming a dummy queue of one element, so receiving less priority. Since more CPU cycles are assigned to source threads, a lower number of cycles remain available to the threads of the other operators yielding higher intra-SPE latencies. However, in all the considered cases, the reduction of the $wait_{external}$ component dominates the slight intra-SPE latency increase, so producing lower e2e latency results and a better user experience. The e2e latency reduction is remarkable: −47% with a rate of 7, 500 tuple/sec compared to the prior version of Lachesis.

5.2.2 Flink results. We run LR on Flink, see Figure 9. Lachesis-Mod pushes the saturation point w.r.t Lachesis, moving it from about 5, 000 tuple/sec to 5, 500 tuple/sec. At the new saturation point, e2e latency is −42% lower than the one with Lachesis while throughput is +4% higher. OS still exhibits poor results with high rates, while EdgeWise cannot be used since it does not support Flink.

5.3 Lachesis-Mod with Real-time Threads

We now focus on the combined effect of the new congestion-aware policy implementation and the new real-time translator. Streaming queries are placed into a control group having a real-time CPU limit of 95% of the overall CPU time, so studying them in a scenario where a limit, though high for performance reasons, is imposed in the CPU demand of the streaming system (e.g., this might be useful to still share computing resources with other running applications/containers in the same Edge/IoT platform).

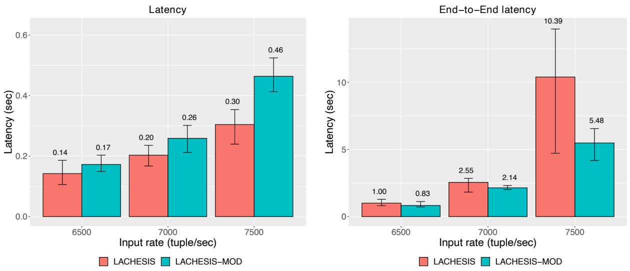

Figure 8: Latency and end-to-end latency of LR on Storm.

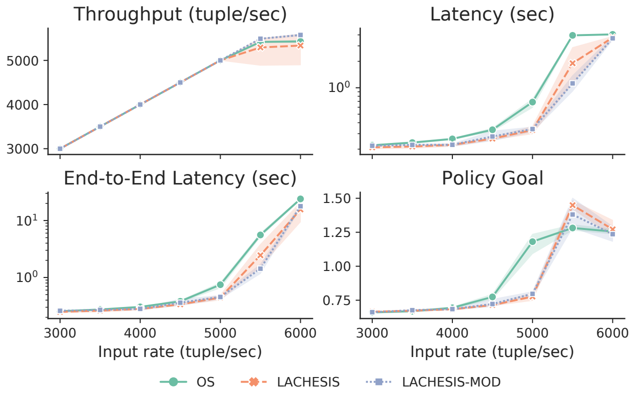

Figure 9: Results of the LR query on Flink.

5.3.1 Storm results. In this section we consider throughput and e2e latency (we have already argued that the minimization of e2e latency is of primary importance in real-world scenarios). For ETL the use of real-time threads guarantees significantly better throughput, see Figure 10. At the maximum considered rate of 2, 250 tuple/sec, the improvement is of +28% compared with Lachesis-Mod. As far as e2e latency is concerned, we observe that the real-time translator allows maintaining the latency below the one of Lachesis-Mod in all considered rates. Although the two curves appear quite close to each other due to the logarithmic scale, the latency reduction is often significant: Lachesis-Mod-RT provides −15% lower e2e latency than Lachesis-Mod at the maximum considered input rate.

Figure 11 shows the results with the STATS query. Although with the highest considered rate of 400 tuple/sec the average throughput is lower with Lachesis-Mod-RT (although this with a higher variability), it performs generally better for all the other rates greater than 350 tuple/sec. The use of the real-time translator produces e2e latency values similar to the one of Lachesis-Mod, although latencies are generally worse with lower input rates in this query.

The use of the real-time translator for the LR query is shown in Figure 12. Lachesis-Mod-RT performs better than Lachesis-Mod

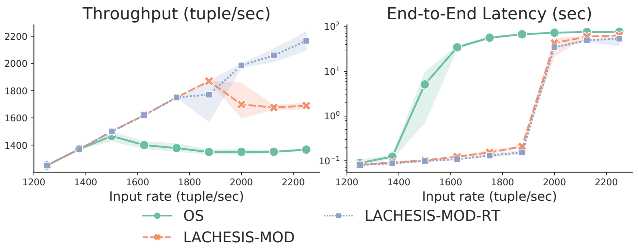

Figure 10: Results of the ETL query on Storm with the real-time translator.

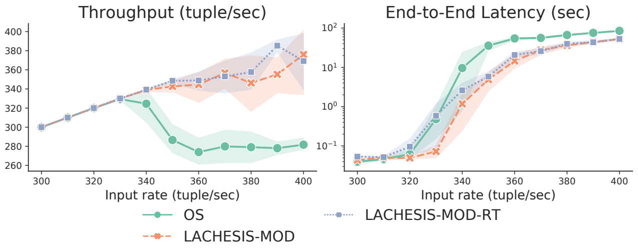

Figure 11: Results of the STATS query on Storm with the real-time translator.

by exhibiting higher throughput (+4% on average) with input rates greater than 6, 000 tuple/sec. In the same range of input rates, e2e latency values are generally lower (−36% on average).

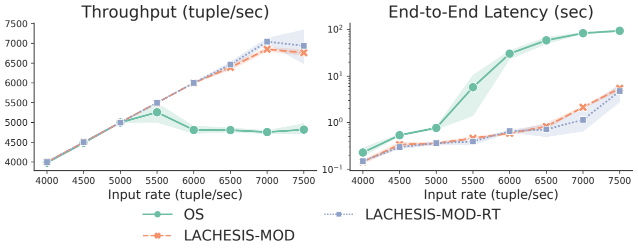

Figure 12: Results of the LR query on Storm with the real-time translator.

5.3.2 Flink results. To complete the analysis, we run the LR query on Flink. Results are shown in Figure 13. In general, the use of real-time threads brings considerable benefits in terms of e2e latency while the improvement is often small in terms of throughput. With the rate of 6, 000 tuple/sec the increase in throughput with Lachesis-Mod-RT is small (few tens of inputs per second). As far as e2e latency is concerned, Lachesis-Mod-RT provides smaller values with all the considered rates compared with Lachesis-Mod.

5.4 Sensitivity Analysis

We analyze the behavior of Lachesis-Mod and Lachesis-Mod-RT concerning the specific priorities applied to threads and the effect on the input queue sizes within the SPE. Furthermore, we study the impact of the scheduling frequency chosen to recompute priorities.

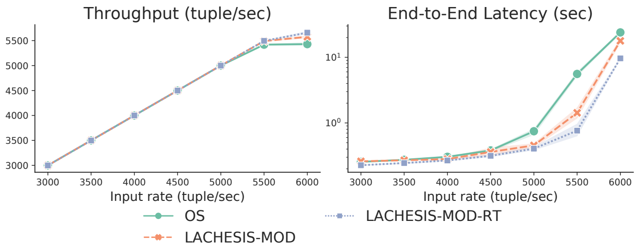

Figure 13: Results of the LR query on Flink with the real-time translator.

5.4.1 Input queue sizes and chosen priorities. We consider for this part the ETL query on Storm. We have ten operators connected in a complex data-flow graph. Fig. 14 shows the average size of the input queues of the operators within Storm during the whole execution duration. Blue bars, corresponding to the default OS execution without Lachesis support, clearly show that some operators are more congested than others. Notably, the SenMlParser, which applies a complex parsing of tuples from raw strings, is the bottleneck having the largest average queue size (slightly less than 600 pending items, see the rightmost bar in the figure). Figure 14 also shows the effect of Lachesis-Mod and Lachesis-Mod-RT. The general effect is that bottleneck operators are more prioritized and receive more CPU cycles for reading inputs from their queues and processing them. Since more congested operator receive more CPU cycles, fewer cycles remain available to other operators whose queue size slightly increases. So, the consequence of applying the congestion-aware policy is that the queues have a more homogeneous size, and this yields better throughput and e2e latency. The better performance exhibited by Lachesis-Mod-RT with real-time priorities in§ 5.3 is justified by the better balancing between queue sizes compared with the use of nice with Lachesis-Mod, as Fig. 14 shows.

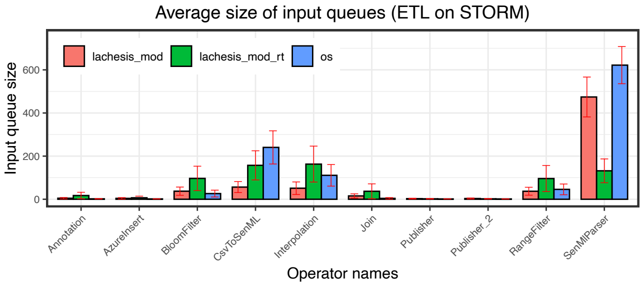

Figure 14: Average size of input queues: ETL query on Storm with input rate of 2, 250 tuple/sec.

Fig. 15 shows the applied priorities. Each bar shows the average priority applied to the main thread of each operator (we omit the priorities of helper threads for the sake of space). In the case of Lachesis-Mod, threads are SCHED_OTHER and priorities are controlled through nice (−20 is the better priority level), while with

Lachesis-Mod-RT threads are promoted in the SCHED_RR class and the highest real-time priority provided by Linux is 99. We can observe that priorities in Fig. 15 map quite easily with the queue sizes in Fig. 14. Operators having larger queues are more prioritized compared with less congested operators. This confirms the right application of the congestion-aware scheduling policy.

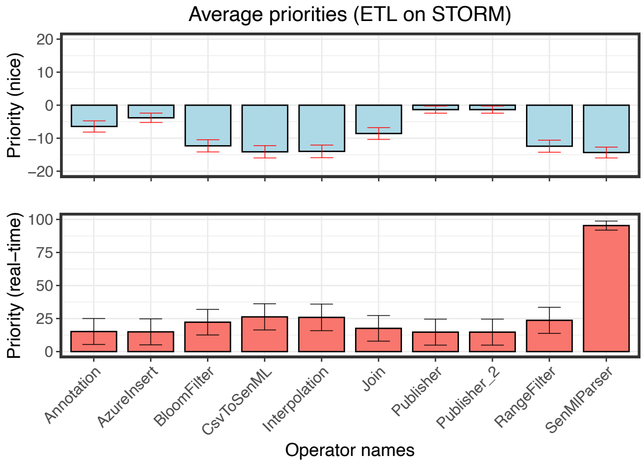

Figure 15: Applied priorities by Lachesis-Mod (top) and Lachesis-Mod-RT (bottom) with ETL query on Storm.

5.4.2 Scheduling frequency. In this part we answer the question related to how frequently Lachesis should re-evaluate the metrics from the SPE and re-compute the thread priorities. In fact, setting a too-slow update of the priorities can impair the effectiveness of our approach, since Lachesis would not be able to prioritize correctly the operators based on their actual congestion. Fig. 16 shows an experiment with the LR query on Flink. We choose this SPE because it provides a finer configuration of how frequently built-in metrics are made available. We evaluate throughput and e2e latency by considering input rates around the saturation limit (see Fig. 13). The analysis considers Lachesis-Mod-RT only.

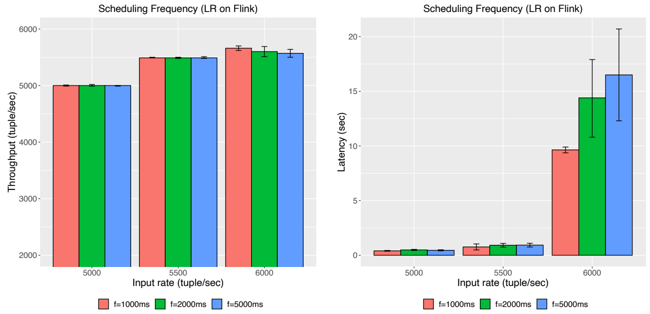

Figure 16: Scheduling frequency (LR query on Flink).

We consider three scenarios, where the main loop of Lachesis is repeated every second (default setting), two seconds and five seconds. As depicted in the figure, throughput is stable and becomes worse when priorities are recomputed every five seconds. This correlates with the effect on e2e latency, which becomes higher in the same conditions, reflecting the fact that with a low frequency our system is not fast enough to assign the right priorities to threads.

In the above considerations, we have to consider that a frequent execution of Lachesis increases the CPU utilization to run the main loop gathering statistics from the SPE and applying priorities to the OS through the chrt command. However, such an overhead is in all cases extremely low: the Lachesis process consumes less than 1% of CPU time with the main loop repeated every second.

5.5 Summary of Results

Tab. 1 highlights the improvements achieved by Lachesis-Mod-RT over the prior version of Lachesis [13], and compared with the default execution with standard OS scheduling. So doing, we provide a quantitative assessment of the significance of the optimizations proposed in this paper: *i)* the refinement of the congestion-aware policy in § 4.1, and the use of the new real-time translator in § 4.2. All values in the table are reported in percentage by considering the average improvement (throughput) and reduction (e2e latency) over the input rates beyond the saturation limit.

		ETL (Storm)	STATS (Storm)	LR (Storm)	LR (Flink)
Lachesis-Mod-RT *vs* Lachesis-Mod	Throughput E2E Latency	+7% −14%	+6.38% −5.49%	+2.49% −11%	+1.1% −27%
Lachesis-Mod-RT *vs* Lachesis	Throughput E2E Latency	+19% −31%	+11% −13%	+3.3% −25%	+2.8% −40%
Lachesis-Mod-RT *vs* OS	Throughput E2E Latency	+38% −71%	+29% −52%	+38% −93%	+5.7% −73%

Table 1: Improvements (percentage) achieved with the optimization/enhancements of Lachesis presented in this paper.

Compared with OS, Lachesis-Mod-RT provides +34.7% throughput and −72% e2e latency on average. Lachesis-Mod-RT exhibits significant improvements w.r.t the already published version of Lachesis in [13]: +9.75% throughput and −27% e2e latency.

6 RELATED WORKS

Scheduling of streaming queries is a research topic born with the first *Data Stream Management Systems* (DSMSs) more than 20 years ago. DSMSs like Aurora [3] had an internal scheduler component capable of running query operators on top of a pool of threads. So, doing scheduling in user space with custom stream-aware policies. Examples of such policies are the First-Come-First-Served (FCFS) policy proposed in [8] to optimize for the maximum latency of streams of continuous requests, the Rate-Based (RB) policy optimizing for the average latency of a single streaming query [17], which has been extended to multiple-query contexts in [14]. A new metric called *mace* (Maximum Cumulative Excess) has been introduced in [10] to accurately estimate the latency of a stream processing pipeline, and the same work proposes a policy based on that metric to schedule operators. The *chain* policy in [7] tries to minimize the runtime memory usage of multiple queries at the same time. The Aurora DBMS has been extended with a two-level hierarchical scheduling policy [9]. The first-level scheduler uses a round-robin policy to schedule queries, while a second-level scheduler prioritizes operators of the query using different policies to optimize throughput (Min-Cost) or average latency (Min-Latency).

Policies addressing the scheduling of multiple queries are designed to take into account fairness. Different metrics have been proposed such as the Longest Stretch First (LSF) metric [4]. Scheduling queries with different priority classes have been explored in [15] with a hierarchical scheduler based on a weighted round-robin query assignment and the Highest Rate policy to minimize latency. These policies need detailed information about the query processing, such as the earliest timestamp of the tuples pending in input buffers of query operators. To extract this information, the scheduler must be tightly coupled with the DSMS runtime system, in order to acquire detailed information in an accurate manner. For this reason, implementing the scheduler in user space, as an internal component of the DSMS, was a promising direction.

In modern SPEs like Storm and Flink, thread scheduling is left to the Operating System, which pursues global goals from the system-wise perspective. Indeed, modern SPEs are designed for large machines, with high availability of CPU cores. When executed on resource-constrained devices such as at the Edge, the solution proposed by EdgeWise [11] was to change the Storm runtime system by adding a user-space scheduler, so proposing again the idea of DSMSs schedulers by adding them to Storm. Lachesis [13] criticizes this approach since it requires intrusive changes to the Storm official code base, it is not portable to newer versions of Storm nor extensible to other SPEs of the same kind (e.g., Flink).

7 CONCLUSIONS AND FUTURE WORKS

In this paper, we proposed different optimizations of Lachesis. We optimized the congestion-aware policy by reflecting more accurately the load of the threads in the SPE, and by assigning priorities to Source operators more accurately. We also showed that the use of the new real-time translator (Lachesis-Mod-RT) brings additional performance benefits. To the best of our knowledge, this was the first attempt of using the real-time features provided by Linux for scheduling streaming queries. These enhancements improve the effectiveness of Lachesis, and demonstrate the effectiveness of real-time Linux threads to accelerate open-source SPEs.

Our work can be further improved in the future. Other threads used by Flink (e.g., the ones within Netty, see Fig. 3) can be controlled through `nice` or `chrt` since they are currently outside the Lachesis control. It would also be interesting to run Storm, Flink and Lachesis with a Linux kernel compiled with the `PREEMPT_RT` patch that minimizes the amount of non-preemptible kernel functions, and so can bring additional performance benefits. In addition, we highlight that the general approach fostered by Lachesis, so to help the Linux scheduler with directives driven by application-specific metrics not accessible by the OS otherwise, can be extended with a broader application to parallel frameworks and task-based parallel runtime systems.

ACKNOWLEDGMENTS

This research has been supported by the INTERCONNECT project no. PRA 2022-2023 9, by the EuroHPC project "Adaptive multi-tier intelligent data manager for Exascale" under grant 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1, and by the National Resilience and Recovery Plan (PNRR) through the National Center for HPC, Big Data and Quantum Computing.

REFERENCES

[1] 2020. Apache Flink. https://flink.apache.org/. [Online; accessed 26-Feb-2020].

[2] 2020. Apache Storm. http://storm.apache.org/. [Online; accessed 26-Feb-2020].

[3] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. 2003. Aurora: A New Model and Architecture for Data Stream Management. *The VLDB Journal* 12, 2 (aug 2003), 120–139. https://doi.org/10.1007/s00778-003-0095-z

[4] Swarup Acharya and S. Muthukrishnan. 1998. Scheduling On-Demand Broadcasts: New Metrics and Algorithms. In *Proceedings of the 4th Annual ACM/IEEE International Conference on Mobile Computing and Networking* (Dallas, Texas, USA) *(MobiCom '98)*. Association for Computing Machinery, New York, NY, USA, 43–54. https://doi.org/10.1145/288235.288248

[5] Henrique C. M. Andrade, Bugra Gedik, and Deepak S. Turaga. 2014. *Fundamentals of Stream Processing: Application Design, Systems, and Analytics* (1st ed.). Cambridge University Press, New York, NY, USA.

[6] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear Road: A Stream Data Management Benchmark. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30* (Toronto, Canada) *(VLDB '04)*. VLDB Endowment, Toronto, Canada, 480–491. http://dl.acm.org/citation.cfm?id=1316689.1316732

[7] Brian Babcock, Shivnath Babu, Rajeev Motwani, and Mayur Datar. 2003. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/872757.872789

[8] Michael A. Bender, Soumen Chakrabarti, and S. Muthukrishnan. 1998. Flow and Stretch Metrics for Scheduling Continuous Job Streams. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (San Francisco, California, USA) *(SODA '98)*. Society for Industrial and Applied Mathematics, USA, 270–279.

[9] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator Scheduling in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 838–849.

[10] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. 2011. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*. 255–266. https://doi.org/10.1109/ICDE.2011.5767926

[11] Xinwei Fu, Talha Ghaffar, James C. Davis, and Dongyoon Lee. 2019. EdgeWise: A Better Stream Processing Engine for the Edge. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 929–946. https://www.usenix.org/conference/atc19/presentation/fu

[12] Wolfgang Mauerer. 2008. *Professional Linux Kernel Architecture.* Wrox Press Ltd., GBR.

[13] Dimitris Palyvos-Giannas, Gabriele Mencagli, Marina Papatriantafilou, and Vincenzo Gulisano. 2021. Lachesis: A Middleware for Customizing OS Scheduling of Stream Processing Queries. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 365–378. https://doi.org/10.1145/3464298.3493407

[14] M.A. Sharaf, P.K. Chrysanthis, and A. Labrinidis. 2005. Preemptive rate-based operator scheduling in a data stream management system. In *The 3rd ACS/IEEE International Conference onComputer Systems and Applications, 2005.* 46–. https://doi.org/10.1109/AICCSA.2005.1387043

[15] Mohamed A. Sharaf, Panos K. Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. 2008. Algorithms and Metrics for Processing Multiple Heterogeneous Continuous Queries. *ACM Trans. Database Syst.* 33, 1, Article 5 (mar 2008), 44 pages. https://doi.org/10.1145/1331904.1331909

[16] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. 2017. RIoTBench: An IoT Benchmark for Distributed Stream Processing Systems. *Concurrency and Computation: Practice and Experience* 29, 21 (2017), e4257. https://doi.org/10.1002/cpe.4257 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.4257

[17] Tolga Urhan and Michael J. Franklin. 2001. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 501–510.