Lachesis: A Middleware for Customizing OS Scheduling of Stream Processing Queries

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

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

Marina Papatriantafilou  
Chalmers University of Technology  
ptrianta@chalmers.se

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

ABSTRACT
Data streaming applications in Cyber-Physical Systems enable high-throughput, low-latency transformations of raw data into value. The performance of such applications, run by Stream Processing Engines (SPEs), can be boosted through custom CPU scheduling. Previous schedulers in the literature require alterations to SPEs to control the scheduling through user-level threads. While such alterations allow for fine-grained control, they hinder the adoption of such schedulers due to the high implementation cost and potential limitations in application semantics (e.g., blocking I/O).

Motivated by the above, we explore the feasibility and benefits of custom scheduling without alterations to SPEs but, instead, by orchestrating the OS scheduler (e.g., using nice and cgroup) to enforce the scheduling goals. We propose Lachesis, a standalone scheduling middleware, decoupled from any specific SPE, that can schedule multiple streaming applications, run in one or many nodes, and possibly multiple SPEs. Our evaluation with real-world and synthetic workloads, several SPEs and hardware setups, shows its benefits over default OS scheduling and other state-of-the-art schedulers: up to 75% higher throughput, and 1130x lower average latency once such SPEs reach their peak processing capacity.

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

KEYWORDS
Scheduling, Stream processing, Middleware

1 INTRODUCTION
Cyber-Physical Systems (CPSs) like Smart Grids and Vehicular Networks are undergoing a data-driven digitization transformation to meet new societal/business goals (e.g., higher penetration of renewable energy or better user experience [30]). This transformation is enabled by (1) the large amounts of data sensed in CPSs, (2) the significant computing power spanning from edge devices to higher-end servers, and (3) analysis paradigms, like data stream processing, which generate value from raw data with high throughput and low latency [18]. Stream processing gained popularity thanks to Stream Processing Engines (SPEs) [3, 9, 32] supporting users with simple ways of defining and deploying applications (called continuous queries, or simply queries). Such queries are defined as Directed Acyclic Graphs (DAG) of streams and operators transforming raw inputs to rich outputs delivered to the end users.

Motivation and Challenges. SPEs can be distinguished as either (1) one at a time, when they process individual inputs (called tuples) as soon as they are available (e.g., Apache Storm [3] and Flink [9]), or (2) microbatched, when they discretize streams into contiguous batches, with each batch being processed as a whole (e.g., Apache Spark [60]). While the latter optimize throughput, with the general aim of approaching the peak memory bandwidth of the underlying machines (often high-end servers [21, 54]), the former are oriented to latency-sensitive applications, and their efficient execution has been analyzed across the entire spectrum of devices found in CPSs [15, 20, 42, 45]. In this work, we focus on one-at-a-time SPEs.

One-at-a-time SPEs (or simply SPEs from now on) execute operators on dedicated per-operator user-level threads, which in most current systems are mapped directly to kernel-level threads and...
are scheduled by the Operating System (OS). While being able to orchestrate many threads concurrently and in parallel, within and across processes [23, 53], OS schedulers are unaware of the specific performance goals of streaming applications. This can lead to suboptimal query performance, especially when CPU resources are scarce, e.g., in resource-constrained edge devices or in multi-tenant scenarios with servers executing queries of many analysts. Custom scheduling, i.e., deciding in which order and for how long operators are executed on the available processors based on SPE-accessible or application-dependent metrics [18, 43, 44, 50], can in such cases significantly improve the performance of queries. Figure 1 outlines the performance benefit of using custom scheduling for a traffic monitoring application from the Linear Road benchmark [4] deployed on edge devices (studied further in §2 and §6). As the input rate increases, custom scheduling significantly improves the average throughput and latency performance of the application in comparison to standard fair scheduling applied by the OS.

To provide custom scheduling, state-of-the-art (SoA) solutions [12, 18, 43] alter the core runtime of SPEs. Instead of having a rigid binding between operators and threads (as in the default runtime of Storm and Flink), they schedule operators entirely in user space as user-level threads. The tight coupling of such User-Level Streaming Schedulers (UL-SS) to the SPE allows them to have fine-grained control over the scheduling decisions. However, the same tight coupling and the associated changes to the SPE can bring implementation and compatibility risks, as evident from the lack of adoption of UL-SS in mainstream SPEs [5, 9, 32]. Additionally, UL-SS miss significant advantages offered by the OS scheduler, such as transparent support for blocking operations (which are not scheduled at all in [18]) and the ability to schedule operators of different SPEs in a homogeneous manner.

**Contribution.** Motivated by the above, we ask: is it beneficial to implement custom scheduling by assisting the OS instead of altering the SPE to rely on a user-level scheduler? We answer affirmatively and introduce Lachesis,1 a middleware for streaming applications that offers fine-grained control of the scheduling decisions without altering the implementation of the SPEs or the queries. Lachesis combines low-level mechanisms of the SPE and the OS to provide high-level scheduling abstractions. More specifically:

1Lachesis is decoupled from SPEs and runs as a separate process, orchestrating the OS scheduler through mechanisms like nice and cgroupl. Since it does not rely on user-level threads, Lachesis is not affected negatively by blocking operations (that would require ad-hoc solutions otherwise [19]). Accounting for the spectrum of CPSs’ devices, Lachesis supports scheduling (1) on low-end devices, scheduling one or more queries running on one device or even distributed by the SPE on more devices, and (2) on higher-end servers, allowing cross-scheduling of multiple different SPEs. To the best of our knowledge, Lachesis is the first middleware with the latter capability. We thoroughly evaluate Lachesis with real-world and synthetic workloads, three SPEs (Apache Flink [9], Apache Storm [3] and Liebre [52]), and both low- and higher-end devices, showing its benefits over default OS scheduling and other state-of-the-art schedulers: up to 75% higher throughput, and 1130x lower average latency once such SPEs reach their peak processing capacity.

We describe data streaming and scheduling in §2, our system model and goals in §3, Lachesis in §4 and §5, our evaluation in §6, related work in §7 and conclude in §8.

### 2 PRELIMINARIES

We now overview data streaming, operator scheduling, and some of the features the OS offers to modify the priorities used to allocate resources to running processes/threads.

A query is a DAG of operators connected by streams (i.e., sequences of tuples sharing a schema). Data Sources, external to the query (e.g., publish-subscribe systems like Apache Kafka [16]) generate ingress tuples usually at varying rates. These ingress tuples are fed to queries by Ingress operators (also called Sources or Spouts [3, 9, 32]). Tuples are pushed through the query operators, possibly resulting in new tuples, and are eventually delivered as egress tuples to Egress operators (also called Sinks [9]), which forward them to the user or other applications/systems. Each operator is characterized by its cost, i.e., the average time to process a tuple, and its selectivity, i.e., the average number of output tuples produced per input tuple. Queries are executed by SPEs [3, 9, 32]. During deployment, SPEs transform the DAG defined by a user, which is known as a logical DAG (or topology) [18] and comprising of logical operators, to a physical DAG, comprising of physical operators, applying optimizations such as operator fusion and fission (usually guided by user-defined configuration parameters) [25]. Physical operators (or simply operators, if not otherwise stated) are the minimum query unit executed on the underlying node.

**In which ways can operators be scheduled?** Our work studies how operators are scheduled inside each node, i.e., which operators are prioritized or given more CPU time.2 Since physical operators are the execution units of SPEs, scheduling boils down to deciding which physical operators are assigned to CPU time. Because SPEs rely on the OS itself for scheduling, they spawn one thread per physical operator [3, 9, 56]. Instead, SoA UL-SSs schedule physical operators as user-level threads which are executed on the hardware by a small number of kernel-level threads [18, 43].

**Example.** Figure 2a shows a simplified view of the traffic monitoring application of Figure 1, illustrating two branches, one computing variable tolls based on the levels of congestion (Branch 1) and one computing a fixed toll (Branch 2) for segments of a set of highways. Assume that branch 1’s operators should have higher priority than those of branch 2 (e.g., to promptly deliver high tolls, indicating congestion, to vehicles approaching busy segments). The logical DAG in Figure 2a could be transformed by the SPE into the physical DAG of figures 2b/c, with operators C, D, E being fused into the same physical operator, and operator F replicated twice.

Figures 2b/c show the query when C/D/E, G and F2 have 1, 1, and 4 tuples in their input queues, respectively, and F2 is the operator that has waited longer to be scheduled. There, the scheduling

---

1In Greek mythology, Lachesis was one of the three Moirai (Fates). She measured the thread of life allotted to each person with her measuring rod.

2The term scheduling is also used sometimes in the literature to refer to the process of deciding where to deploy operators onto distributed nodes [2, 10, 34, 57–59]. Our work is orthogonal to such job placement techniques.
Figure 2: Simplified overview of the monitoring query from §1 (a), deployed with/without a custom scheduler (b/c).

decision that prioritizes branch 1 is to schedule C/D/E and G. A UL-SS, able to observe the internal state of each queue and aware of the scheduling goal [18], can dispatch C/D/E and G ignoring that F2 has not run for longer (2b). In that case, while the OS chooses the kernel-level thread that is given access to the CPU (bottom row of switches in the figure), the UL-SS decides which operator (i.e., user-level thread) should be executed (top row of switches). On the other hand, if scheduling is left to the OS, its common fairness goal [35] and its lack of awareness of the user’s scheduling preferences can result in higher chances of running F2 (2c).

How can the OS scheduling be customized? Modern OSs offer mechanisms to customize their internal scheduling but lack easy ways for users to express arbitrary, fine-grained scheduling policies. Here, we focus on Linux, whose open-source APIs and widespread use in edge CPS’ devices and higher-end servers make it an ideal candidate for scheduling customizations. While not aiming at providing an exhaustive discussion about all such mechanisms of Linux, we focus on two that, together, as we show in this work, can be used in a complementary fashion to enable Lachesis.

Thread Niceness. The Linux scheduler maintains a list of queues, one for each priority value. Normally, all threads belong to the queue with priority zero (the lowest, of normal, non-real-time threads). Within this queue, threads are kept ordered based on the vruntime parameter, which represents the actual time spent in execution by the thread, weighted by the whole load of the queue. The OS continuously updates the vruntime on a per-thread basis and schedules threads with minimum vruntime by preempting running ones if needed. This ordering can be partially controlled with the nice command, which assigns an integer in the interval \([-20, 19]\) to a thread. Each nice value \(n\) is statically mapped to a constant weight \(w(n) = \frac{1024}{1+25^n}\). Thus, the ratio between the weights of two threads with nice of \(n_1\) and \(n_2\) is computed as \(1.25^{n_1-n_2}\). At a high level, an increase in the nice value reflects in an increase of the vruntime parameter, decreasing the probability for a thread to run (and vice-versa). Although nice can effectively control thread (and thus operator) priorities, it only supports 40 distinct values, indicating a need for additional mechanisms in multi-query setups.

Control Groups. Control groups [22] (cgroups) are an alternative way to control resources such as CPU time (the focus of this work). They are groups of threads constructed as hierarchies rooted at one or more resource controllers (also called subsystems). The scheduling of threads in each cgroup is based on their relative nice values in that group (i.e., the whole range of nice can be used in each cgroup without interference from other processes). The CPU controller allows the control of the CPU time allocated to the threads of each cgroup through the cpu.shares parameter, which defines the relative share of CPU time for all the threads in that cgroup. For example, if we split the threads running a set of physical operators evenly into two cgroups with equal cpu.shares, we can ensure that the processing power available to the two subsets will be balanced. In the extreme case, it is possible to assign each thread to a dedicated cgroup to gain further control of their runtime.

3 SYSTEM MODEL AND GOALS

We study how to schedule a set of operators, from one or more queries, executed in one or several nodes and with possibly multiple, different SPEs, without modifications to the runtime system of such SPEs. Each query is run in one SPE that can be composed of one or more processes distributed on the underlying nodes, which are dedicated to stream processing queries. We consider one-at-a-time SPEs like Storm [3], Flink [9], and Liebre [52], where each physical operator is executed by a dedicated thread. We study the mainstream scenario where each SPE exposes quantitative information about the running queries and operators through public APIs (e.g., for debugging and monitoring purposes). Furthermore, we assume that each SPE process is running on a machine equipped with a Linux distribution offering thread nice and cgroup mechanisms. Below we define auxiliary terms, relevant to our problem statement.

The term entity refers to logical/physical operators/threads when discussing aspects that apply to all such concepts.

Definition 3.1 (Metric). A metric is a triplet \((r, e, value)\) providing quantitative information about entity \(e\) at time \(r\).

Metrics can be primitive or derived as transformations on primitive metrics on which they depend.

Definition 3.2 (Scheduling Policy). A scheduling policy is an algorithm receiving a set of metrics (sharing time \(r\)) and outputting priority values for a set of physical operators.

The priority given to a physical operator must be translated into an input for the scheduling mechanism of the underlying OS, which is handled by a translation policy.

Definition 3.3 (Translation Policy). A translation policy is a strategy to translate priority values obtained from a scheduling policy into OS-related parameters (i.e., nice or cpu.shares of cgroup) associated with the corresponding threads.

---

3 SPEs can spawn additional helper threads e.g., in charge of copying/serializing tuples [3, 9]. Lachesis can control their priority similarly to physical operators, so, for brevity, we do not discuss them as separate entities.
3.1 Problem Definition and Goals

We want to define a middleware that runs separately from the SPEs and uses metrics to apply a broad spectrum of scheduling policies by tuning the behavior of the Linux scheduler without changing the implementation of the SPEs or the queries. To have real-world value, that middleware should be able to:

- Enforce arbitrary, user-defined scheduling policies, taking advantage of appropriate OS scheduling mechanisms. Enforce such policies on different SPEs without alteration. Schedule multiple queries at a time, possibly optimizing different goals for each query. Schedule queries deployed in different SPE processes running on multiple nodes. Achieve G3 also for queries running on different SPEs.

3.2 Performance Metrics

Here, we introduce in more detail common metrics used to evaluate the performance of streaming queries [18, 43, 51].

<table>
<thead>
<tr>
<th>Metric Provider</th>
<th>Algorithm 1: Main loop of Lachesis.</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Input</td>
<td>T[K]: User-selected translators (one for each policy)</td>
</tr>
<tr>
<td>Lachesis Data</td>
<td>D[N]: SPE Drivers (for N distinct SPEs)</td>
</tr>
<tr>
<td>ÌD: Metric provider</td>
<td>// Each policy has its own period and required metrics</td>
</tr>
<tr>
<td></td>
<td>ÌD.register({ pmetrics }) // Register required metrics p∈P</td>
</tr>
<tr>
<td></td>
<td>while True do // Main loop start</td>
</tr>
<tr>
<td></td>
<td>if 3p ∈ P: p.timeToRun() then // Run scheduler</td>
</tr>
<tr>
<td></td>
<td>ÌD.update(D) // Compute metrics, Algorithm 3</td>
</tr>
<tr>
<td></td>
<td>for i ∈ [0, K) do // Each policy and translator</td>
</tr>
<tr>
<td></td>
<td>if F[i].timeToRun() then</td>
</tr>
<tr>
<td></td>
<td>schedule ← F[i].schedule(ÌD, D)</td>
</tr>
<tr>
<td></td>
<td>T[i].apply(schedule) // OS mechanism</td>
</tr>
<tr>
<td></td>
<td>sleep(GCD( \bigcup p.period)) // Until time to check again</td>
</tr>
</tbody>
</table>

Figure 3: The architecture of Lachesis.

- Throughput: #tuples ingested by an operator per time unit. 
- Processing Latency (or simply latency): time interval between the output of an egress tuple t and the time when all the ingress tuples that contribute to t were ingested by the Ingress operator(s).
- End-to-end Latency: time interval between the output of an egress tuple t at the sink and the time when all the ingress tuples that contribute to t were produced by the Data Source(s).
- CPU Utilization: percentage of the total CPU physical operators utilize, across all available processors (0-100%).

When the SPE comes close to saturation, an increase in the end-to-end latency (rather than the processing latency) is often beneficial for Data Sources to adapt their rates (e.g., through back-pressure) [45, 51]. For measuring latencies in distributed setups, we assume synchronized clocks (e.g., using NTP [37] or PTP [17]).

4 ARCHITECTURE

In this section, we describe Lachesis’ architecture, outlined in Figure 3, in relation to the goals of §3.1. Users configure Lachesis through the Scheduler UI, choosing the scheduling and translation policies. During Lachesis’ main loop, shown in Algorithm 1 and explained in the following, the policies are executed at regular scheduling periods, retrieving runtime information from the SPE drivers and the metric provider, computing a new schedule (i.e., priorities of the physical operators) and applying it with translators. The latter implement translation policies using specific OS mechanisms. Below, we describe each component in more detail.

**SPE Drivers.** An SPE driver acts as a bridge between the SPE process(es) and Lachesis by pulling runtime information from public APIs of the SPEs (cf. §3) without altering any part of the SPE implementation. To enable transparent support for multiple SPEs (G2, §3.1), drivers convert low-level runtime data to information about entities (physical operators, logical operators, and threads), hiding SPE-specific details and allowing the other components of Lachesis to work at an abstract level. Drivers are also responsible for exposing raw metric data retrieved by SPEs and other external systems to the metric provider. The use of public APIs simplifies driver implementation and lowers the adoption risk of custom scheduling. Such external drivers allow Lachesis to run as a separate process without the potential to introduce bugs into the SPE. Furthermore, the drivers reduce the maintenance effort of custom scheduling since public APIs evolve slowly and maintain backward compatibility. For example, in §6 Lachesis uses the same driver to schedule different versions of Storm, while a UL-SS would require a manual port to the newer version.

**Metric Provider.** Lachesis comes with an extensible set of metrics, each of which defines (1) its dependencies on other metrics and (2) a function to compute its value from those dependencies. The metric provider is the single entity responsible for computing the metrics requested by the policies by fetching and transforming raw metrics provided by each SPE driver (Algorithm 1 L1, L4) and exposing them to the policies (L7). Different SPEs expose different raw metrics, and the metric provider is tasked with using the dependency information to compute the requested values (as discussed in §5.2).

**Translators.** Lachesis’ policies compute a schedule as priorities of the (physical) operators. A translator uses an OS scheduling mechanism to apply that schedule after converting it to a format appropriate for that mechanism (Algorithm 1, L8). Since translators are orthogonal to the policies, it is possible to change the translator (and thus the underlying OS mechanism) without changing the
policy itself. For example, *Lachesis* could switch from using nice to using other OS mechanisms (e.g., cpu.shares, used in §5 and §6).

Scheduling Policies. A scheduling policy computes a schedule using a set of entities from the SPE drivers and a set of metrics from the metric provider (L7). To schedule multiple queries, possibly with different strategies, potentially executed by different SPEs (G3 and G5, §3.1), *Lachesis* can use multiple policies (e.g., one policy per query), each with its own period (L3, L6). This, combined with each policy being able to use a different translator (i.e., OS mechanism), makes it possible to have fine-grained, multi-dimensional scheduling decisions, as we show in §6. Furthermore, it allows *Lachesis* to switch scheduling policies at runtime (by enabling one policy and disabling another), with the conditions of this switch programmed by the user. To account for policies with different periods, *Lachesis* wakes up at the minimum time interval between policy invocations (L9) but only runs if there is at least one policy to execute.

The decoupled architecture of *Lachesis* allows policies to be abstract without the need to consider the characteristics of a certain SPE or OS mechanism. Consequently, users can develop implementations of arbitrary policies only once and then reuse them to schedule operators in any SPE supported by *Lachesis*, as further discussed in §5 and §6.

5 DESIGN AND IMPLEMENTATION

5.1 Specifying User-Defined Scheduling Goals

SPEs convert the logical DAGs to physical ones during query deployment (cf. §2). Even though scheduling policies refer to physical operators (Definition 3.2), users might want to express their scheduling preferences independently of how DAGs are converted from logical to physical, i.e., in terms of logical operators. Such policy definitions can enable policy reuse in different deployments and SPEs (G2 and G3, §3.1). With this consideration, *Lachesis* allows users to either (1) define a policy directly for physical operators, (when they know the structure of the physical query DAG), or (2) define a policy in a decoupled fashion, combining a high-level policy that refers to logical operators and a reusable transformation rule that converts the logical schedule to a physical one.

A sample transformation rule is shown in Algorithm 2. This rule takes the schedule of a high-level policy as input and produces a physical schedule where the priority of fused physical operators is the highest priority of the logical operators that comprise them. *Lachesis* already comes with such basic transformation rules which are used in our evaluation.

To show that *Lachesis* can accommodate various high-level scheduling policies, we present (and evaluate in §6) four such policies, commonly used in the literature [18, 43]:

| Queue Size (QS) [18] prioritizes operators with more input tuples in their queues, balancing such queues’ size and, in turn, the operators’ effective utilization, to achieve higher throughput at the Egresses and to lower latency. 
Highest Rate (HR) [50] prioritizes “operator paths” (branches of a DAG ending at a sink) that are both productive (i.e., with operators having high selectivity) and inexpensive (low cost) with the goal of minimizing the average processing latency of all the tuples in the system. First-Come-First-Serve (FCFS) [7] prioritizes those operators whose input tuples have spent more time in the system, with the goal of minimizing the maximum latency. RANDOM gives operators uniformly random priorities.

5.2 Offering SPE-Agnostic Metrics

As mentioned in §4, each metric in *Lachesis* declares its dependencies, forming a directed, acyclic dependency graph rooted at that metric. Note that different SPEs might have only some parts of such a graph available. For example, Figure 4 shows an example graph for the goal of the HR policy, expressed as a metric. Notice that none of the two example SPEs exposes the desired metric directly. Hence, to keep scheduling policies independent of SPE-specific details (G2, §3.1), the metric provider needs to compute the metric based on its dependencies using a separate strategy for each SPE.

Algorithm 3 shows how *Lachesis* achieves the above. In the update method, called at each scheduling period, the metric provider iterates through all drivers (i.e., for the SPEs being scheduled) and uses compute (L6) to compute all registered metrics for each SPE. The compute method relies on a per-driver cache (L4, L10–11) to prevent duplicate computations of the same metric in each period. If the metric has not been computed yet, *Lachesis* first tries to fetch it directly from the SPE driver (L12–13). If that fails, then the metric provider calls compute recursively to traverse the dependency graph (L16) before computing the metric from its dependencies, saving it in the cache, and returning it (L17–18). Looking back at the example of Figure 4, for SPE A, *Lachesis* would compute the “Path Selectivity” using the “Operator Selectivity”, and then combine the former with the “Path Cost” to compute the “Highest Rate”. For
5.3 Enforcing Policy Priorities

Lachesis’ translators control the OS mechanisms that enforce the user-defined schedule. One such mechanism is nice, which offers thread-specific priorities but, as discussed in §2, offers only one scheduling dimension and allows for limited control (i.e., only 40 distinct priority values). To overcome this limitation, Lachesis also takes advantage of the cgroup mechanism, since it allows for sets of threads/operators to be given different priorities using features such as cpu.shares. Furthermore, when threads are placed in a cgroup, their nice values affect only threads in the same cgroup, opening up the possibility for effective multi-dimensional scheduling (e.g., for multiple queries or multiple branches of a single query, G3 in §3.1). Lachesis accounts for these aspects by defining two complementary formats for schedules:

The single-priority schedule gives a numerical priority to each (physical) operator. It is defined as a dictionary \( \{ \text{tid} \} \rightarrow \mathbb{R} \) from thread IDs to real-numbered priorities. The grouping schedule describes how to assign operators to groups and the priority of each group. It is a dictionary \( \{ \text{gid} \} \rightarrow \{ \mathbb{R}, \text{tid} \} \), mapping the group IDs to tuples containing the group priority and a set of thread IDs that belong to that group.

According to these two formats, Lachesis defines a nice translator for single-priority schedules, controlling the niceness of threads based on the schedule’s priorities, and a cpu.shares translator for grouping schedules, assigning each group of threads to a cgroup and controlling the relative share of CPU time given to each cgroup based on the schedule’s priorities. These translators can be used on their own or combined to offer more scheduling dimensions. For example, in the query of Figure 2, the operators of each branch could be split into two groups, with the cpu.shares translator controlling the relative CPU time available to each branch and the nice translator prioritizing operators inside each group.

While schedule priorities are real numbers, the OS usually expects discrete priorities in specific ranges, e.g., integers in \([-20, +19]\) for nice. To hide such details from the policies, keeping them general and independent of the OS mechanism (G1, §3.1), Lachesis’ translators perform priority normalization, converting the policy priorities to the appropriate numerical units. This is done by a normalization function \( F \), whose type depends on the combination of used policy and translator. For example, a policy that produces integer priorities in the range \([-20, +19]\) might not need special normalization when used with nice but will need one when used with cpu.shares. For policies with linear priorities (e.g., QS), Lachesis uses min-max normalization and discretizes priorities to the required interval. For logarithmically-spaced priorities (e.g., in HR [50]) Lachesis uses min-max normalization on the logarithms of the priorities. For nice, the interpretation of the OS priorities is known and equal to \( p_1/p_2 = 1.25^{n_2−n_1} \) (see §2). Since \( n_{\text{max}} \) is also known (e.g., \(-20\)), we set \( p_1 = p_{\text{max}} = \max(p_i) \) and compute all \( \text{nice values} \) as \( F(x) = n_{\text{max}} \frac{\log(p_{\text{max}}) − \log(x)}{\log(1.25)} \). As nice values are bounded in the range \([-20, +19]\), an additional min-max normalization might still be required if the relative difference between \( p_{\text{min}} \) and \( p_{\text{max}} \) is too big to fit in the range.

6 EVALUATION

In this section, we illustrate Lachesis’ performance benefits compared to SoA UL-SSs and the default OS scheduling (or simply OS) in various streaming application deployments. As previously discussed, CPS can utilize both resource constrained edge devices but also more powerful servers. Following this spectrum, we evaluate Lachesis both in lower-power devices, comparing with the SoA in single-query (§6.2, §6.3), multi-query (§6.4), and distributed scale-out deployments (§6.5), and in a multi-tenancy scenario, with a higher-end server running multiple queries and SPEs concurrently (§6.6). Table 1 outlines all experiment configurations (made available at [40, 41]) and the performance highlights of Lachesis.

6.1 Evaluation Setup

Hardware and Software. We use (1) Odroid-XU4 [24] devices (or simply Odroid), similar in power to edge CPS devices [18, 43], mounting Samsung Exynos5422 Cortex-A15 2Ghz and Cortex-A7 Octa core CPUs, 2 GB RAM and (2) a higher-end server, mounting Intel Xeon E5-2637 V4 @ 3.50GHz (4 cores, 8 threads), 64 GB RAM. All devices run Ubuntu 18.04 and OpenJDK 1.8.0. Node clocks are synchronized with NTP [37] in the local network.4 Lachesis retrieves SPE metrics from Graphite [13], which is supported by all evaluated SPEs. Graphite allows Lachesis to collect metrics with a minimum time resolution of one second. Thus, in this evaluation, Lachesis’ scheduling period is set to one second (i.e., metrics are fetched and decisions are recomputed with this period) and is sufficient, in most cases, to outperform SoA while allowing Lachesis to keep a low resource footprint (discussed in the last part of the evaluation). Except for §6.5 and §6.6, all processes run in a single Odroid, with the SPEs running on the big cores. Experiments are at least 10 minutes long and repeated at least 5 times, similarly to [18, 43]. The data is averaged after discarding the warmup and cooldown.

---

4 NTP is adequate for the evaluated setups, as the latency differences are in the magnitude of tens of milliseconds or more, i.e., an order of magnitude higher than any clock skew.
**Table 1: Summary of the configurations explored in the evaluation.**

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Baseline</th>
<th>Goals (see §3.1)</th>
<th>Queries</th>
<th>SPEs</th>
<th>Policies (see §5.1)</th>
<th>Translators (see §5.3)</th>
<th>Figures</th>
<th>Highlights (compared to the baseline)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-Query (Odroid) §6.3</td>
<td>OS</td>
<td>G1, G2</td>
<td>STATS</td>
<td>Storm, Flink [9]</td>
<td>QS, RANDOM</td>
<td>nice</td>
<td>9, 10, 11, 12</td>
<td>+ 75% throughput</td>
</tr>
<tr>
<td>Multiple Queries (Odroid) §6.4</td>
<td>Haren [43]</td>
<td>G3</td>
<td>SYN</td>
<td>Liebre [52]</td>
<td>QS, FCFS, HR</td>
<td>cpu.shares</td>
<td>14, 15, 16</td>
<td>+ 1130x latency</td>
</tr>
<tr>
<td>Scale-Out (1-4 Odroids) §6.5</td>
<td>OS</td>
<td>G4</td>
<td>LR</td>
<td>Storm, Flink</td>
<td>QS</td>
<td>nice</td>
<td>17</td>
<td>+ 43% throughput</td>
</tr>
<tr>
<td>Multiple Queries &amp; SPEs (Server) §6.6</td>
<td>OS</td>
<td>G5</td>
<td>VS + LR</td>
<td>Storm + Flink + SYN</td>
<td>QS, nice + cpu.shares</td>
<td>18</td>
<td></td>
<td>+ 60% throughput</td>
</tr>
</tbody>
</table>

Average values. As shown in the relevant figures, in some comparisons the baseline has saturated whereas Lachesis has not, amplifying the latency improvement.

**Queries.** We use five different queries in this evaluation:
- [leftmargin=*]An Extract-Transform-Load (ETL) query, part of the RIoTBench suite [51], which reads a stream of IoT sensor data, filters outliers, interpolates missing values, and adds extra annotations to the data; used previously to evaluate EdgeWise [18], composed of 10 operators. **STATS**, another query from RIoTBench that performs three kinds of statistical analytics; also used previously to evaluate EdgeWise; composed of 10 operators. We refer the reader to [51] for more details on ETL and STATS. **Linear Road (LR)**, which includes queries adapted from the Linear Road benchmark in [61, 62], an established streaming benchmark simulating a tolling system for motor vehicle expressways, introduced in [4]. **VS**; a simplified DAG of LR is shown in Figure 2. **VoIPStream (VS)**, a query from the DSPBench benchmark [8] that analyzes call detail records to detect telemarketing users using Bloom filters. Composed of 15 operators making intensive use of group-by distributions. A set of 20 **Synthetic (SYN)** queries, each a pipeline of 5 operators generating a synthetic CPU load per tuple. Each query has a uniformly random cost and selectivity, as in [43, 49]; used previously to evaluate Haren [43]; can simulate blocking operations to assess the impact of I/O.

ETL, STATS use the input data from the EdgeWise paper [18]. Input data for LR, VS is generated with the corresponding benchmark suites. SYN inputs are generated on the fly.

**Baselines, Scheduling Policies, and Translators.** In all experiments, we evaluate the default OS scheduling (OS) and **Lachesis**. Our query setups and baseline UL-SSs are chosen to match the ones evaluated in related works, considering only OS in the absence of such a UL-SS. Our baseline for ETL and STATS is the UL-SS EdgeWise [18], for LR and VS the OS, and for SYN the UL-SS Haren [43]. Unless otherwise stated, both EdgeWise and Haren use the best configuration described in their publications [18, 43]. For **Lachesis**, we evaluate the policies of §5, ranging between 15 (FCFS) to 150 (HR) lines of code and applied using the nice and cpu.shares translators. To ensure access to a common set of resources in all experiments, Lachesis nests the SPE threads under a custom root cgroup.

**SPEs.** We evaluate Lachesis in three SPEs: ETL and STATS queries on Apache Storm 1.1.0 (the version compatible with EdgeWise), LR and VS on Storm 1.2.3 (the latest version that runs out-of-the-box on Odroids) and Apache Flink 1.11.2 [9], and SYN on Liebre 0.1.2 [52]. We had to write approximately 350, 220, and 250 lines of code for the SPE Drivers of Flink, Storm, and Liebre, respectively.

**Data Sources.** The Data Sources replay existing input traces, allowing to run experiments with increasing input rates (until queries saturate) similarly to previous work [18]. The Data Sources are Kafka producers on a different device than the queries. The only exceptions are the ETL and STATS queries, where we generate data in a separate thread exactly as in the original EdgeWise evaluation [18], to have a fair comparison.

**Metrics and Performance Behavior.** We evaluate using metrics from §3.2 and, more specifically, the sum of throughputs of all Ingress operators, as well as the average latency and end-to-end latency over all Egress operators. We also present the values of the goal for each evaluated scheduling policy. Unless otherwise stated, the metric values are averages over time. We study the whole latency distribution separately (cf. § 6.3.1). In any SPE, as input rates (and thus the load) increase, throughput increases accordingly until a saturation point, at which it plateaus (and possibly decreases). End-to-end latency gradually increases until the saturation point. For higher rates, it explodes and keeps growing, as the queue of tuples from the data source to the Ingress grows unbounded [18]. Better system behavior results in saturation at a higher input rate. The behavior of the processing latency depends on intra-query queuing delays which in turn depend on the scheduling decisions.

### 6.2 Can Lachesis Perform Better than the SoA in Single-Query Scheduling?

Here, we compare Lachesis to EdgeWise in single-query scheduling, the scenario evaluated in the original publication [18]. Figure 5 shows the performance of ETL when scheduled either with the default OS scheduling (OS), Lachesis with the QS policy, or EdgeWise with the same policy. Each line shows the mean metric values for each experiment and input rate, with shaded areas (for all the graphs in this section) representing the 95% confidence interval across repetitions. As shown, Lachesis outperforms in throughput both OS (+18%) and EdgeWise (+8%), keeping up with the external rate up to 1625 t/s, in contrast with 1375 t/s for OS and 1500 t/s for EdgeWise. Just before saturation (1625 t/s), Lachesis achieves 50x lower latency than OS and 92x lower latency than EdgeWise as well.
as 65x and 133x lower end-to-end latency, respectively. The QS policy goal (i.e., minimization of the variance in the sizes of the input queues) is optimized by Lachesis significantly better than OS and similarly to EdgeWise, up to the saturation point. Figure 6 shows the distribution of queue sizes for the three scheduling methods. Lachesis achieves small, homogeneous input queue sizes until the saturation point, in contrast with OS, which allows some queues to grow even for low input rates, leading to performance degradation. EdgeWise keeps the queues more homogeneous than OS but leads to all queues settling to a higher size for input rates above 1625 t/s, explaining the smaller performance gain compared to ETL. This high variance in queue sizes is due to a single bottleneck operator which, however, is orthogonal to scheduling and can be used in synergy with both UL-SS and Lachesis.

6.3 Can Lachesis Perform Better than the OS in Single-Query Scheduling for Other SPEs?

Here, we compare OS and Lachesis’ performance in a more recent version of Storm (v1.2.3), and Flink for LR and VS. Since no UL-SS exists for such SPEs,5 we include the RANDOM policy, similarly to [18], to show that the performance improvements of Lachesis are not just a consequence of altering OS thread priorities arbitrarily.

Storm. Figure 9 outlines the performance of Lachesis for the LR query. Lachesis achieves 30% higher throughput than OS (6500 t/s compared to 5000 t/s for OS) and to 200x lower latency (at 6500 t/s). Lachesis also leads to much lower end-to-end latency, up to 34x lower than OS (at 6500 t/s). As expected, RANDOM behaves equally bad to OS. This is also illustrated when observing the scheduling goal of QS, which is consistently lower for Lachesis. Figure 10 shows the performance of the VS query for the same setup. Lachesis manages to sustain a rate up to 3500 t/s, in contrast with OS that can sustain only up to 2000 t/s (+75% improvement). Furthermore,

---

5EdgeWise is bound to an old version of Storm, 1.1.0.
Lachesis attains up to 1130x lower latency and up to 923x lower end-to-end latency (at 3500 t/s). When Lachesis is used, the scheduling goal remains low for all experiments.

Flink. We begin with LR, shown in Figure 11. Lachesis achieves slightly higher throughput than OS, along with up to 9x lower latency and 6x lower end-to-end latency (at 5500 t/s). The goal is optimized by Lachesis until the saturation point where a bottleneck operator prevents custom scheduling from minimizing the queue sizes, similarly to STATS (§6.2). As in LR, the performance of VS in Flink, shown in Figure 12, is lower than in Storm. Although the query quickly saturates, Lachesis improves the scheduling goal and attains up to 38% lower latency than OS. In contrast to Storm, Flink keeps operator queues bounded, regardless of the scheduling. Flink’s effective backpressure results in a smaller variance of queue sizes (reflected by smaller values of the QS goal for the same queries compared to Storm), and thus a smaller potential for the QS policy to further improve the query performance in this SPE.

6.3.1 Does Lachesis improve the tail latency? As shown above, Lachesis can significantly improve average query latency. Although related works focus on such average [18], tail latency can also be important, especially in interactive or large-scale streaming applications where even short-term latency spikes can severely affect the performance of a whole system [14, 33]. For this reason, we study here the whole latency distributions of the query setups of §6.3. We visualize these distributions as letter-value (or boxen) plots, an extension of boxplots that includes more information about the tails of distributions [26]. Letter-value plots replace boxplot whiskers with a variable number of letter-values (LVs), which represent quantiles. The number of displayed LVs depends on the input data. An LV plot begins with the median line (first LV), expanding upward and downward with a pair of boxes that contain 25% of the data, etc.) and the corresponding boxes have decreasing widths.

Figure 13 shows the letter-value plots for the latencies of the four queries studied above. As seen in the plots, Lachesis’s scheduling generally improves not only the average but also the tail

---

6 Chaining (i.e., fusion) is disabled in Flink to have the same physical DAGs as in Storm, but the performance trends are similar when chaining is active.

7 The end-to-end latency distributions, not shown here, follow similar trends.
Figure 13: Letter-value (boxen) plots of the latency distributions of LR/VS in Storm/Flink.

Latency, compared to OS. More specifically, for LR in Storm, Lachesis achieves 79x lower 99th percentile latency and 44x lower 99.9th percentile latency compared to the OS (at 6500 t/s). Similarly, for VS in Storm, Lachesis reduces the 99th percentile latency by up to 358x and the 99.9th percentile by 215x (at 3500 t/s), following the trend of the average latency seen in Figure 10. For LR in Flink, Lachesis reduces the 99th and 99.9th latency percentiles by approximately 2x (at 5500 t/s). Lastly, for VS in Flink, though Lachesis pulls the latency distribution downwards, it leads to slightly higher values in the upper percentiles, at worst increasing the values of the 99th and 99.9th latency percentiles by approximately 39% (at 3000 t/s).

6.4 Can Lachesis Perform Better than the SoA When Scheduling Multiple Queries, Possibly with Blocking Operators?

We now evaluate SYN queries in Liebre [52] using three scheduling policies, comparing Lachesis and the UL-SS Haren, as in the latter’s original evaluation [43]. This experiment deploys 100 operators, but nice has only 40 distinct priorities (cf. §2), so we use cpu.shares translation, assigning each operator to its own cgroup.

The results, in Figure 14, include the scheduling goals of all policies we evaluate. The color of each line represents the scheduler, and the line style the scheduling policy. As shown, Lachesis’ performance is between Haren and the OS for most of the metrics. Comparing Lachesis with OS, we observe that QS and FCFS succeed in keeping the queue sizes small and thus improve the overall system throughput (up to 12%). The latency is significantly reduced (up to 25x at 1000 t/s) and the end-to-end latency exhibits a big drop (up to 66x at 1250 t/s). The HR policy mostly achieves its goal, albeit with a smaller improvement over the OS (up to 42% lower average tuple latency). Note that HR is the only evaluated policy that does not react directly to the metric it tries to optimize (average tuple latency), but instead attempts to optimize it indirectly based on the operator cost and selectivity. While outperforming EdgeWise (§6.2),

Figure 14: Multi-query scheduling of SYN in Liebre.

Figure 15: The effect of scheduling granularity on Haren.
we see here that Lachesis performs worst than Haren. It should be noted, though, that in this setup Haren takes 20x more scheduling decisions than Lachesis (every 50 ms rather than every 1 s), with finer-grained access to fresh, accurate metrics from within the (coupled) SPE runtime itself. In fact, as shown in Figure 15, when using the same scheduling period of Lachesis (HAREN-1000), Haren’s performance becomes comparable (Throughput and End-to-end Latency) or worse (Latency and FCFS goal) than Lachesis’.

How does Lachesis deal with blocking? As discussed in §1, a drawback of UL-SS is that they cannot transparently handle operators that block (i.e., for I/O). Since Lachesis relies on the OS scheduling mechanisms, it is unaffected by this issue. This is shown in Figure 16, with the same setup as Figure 14 (picking only FCFS) but with a random set of 10% of the operators having at 0.1% chance to block for up to 200 ms every time they process a tuple (i.e., simulating an I/O operation such as committing to a remote system). In this case, Lachesis outperforms Haren with up to 43% higher throughput, up to 4.5x lower latency (at 1125 t/s) and up to 331x lower end-to-end latency, while it achieves similar values for the policy goal.

6.5 Is Lachesis Beneficial When Scaling Out?
In this experiment, we explore whether Lachesis can work equally well in scale-out, distributed query deployments. We run LR either in Storm or Flink (not concurrently), increasing the fission degree (parallelism) of all operators of the query from 1 to 2 and 4 and deploying the operators to an equal number of Odroids, each of which runs a separate instance of Lachesis. The different instances of Lachesis run independently, without communicating with each other. The results are shown in Figure 17 for the QS policy. Lachesis follows the same trend as in the non-distributed experiments, illustrating that, in this case, even isolated scheduler instances without global knowledge can bring significant performance benefits. In Storm, Lachesis attains higher throughput (up to 31% at input rate 25000 t/s) and up to 12x lower latency and end-to-end latency (at 11000 t/s). In Flink, Lachesis reaches 10x lower latency (at 11000 t/s) and 7x lower end-to-end latency (at 11000 t/s).

6.6 Is Lachesis’ Multi-SPE Scheduling Beneficial?
Lachesis is the first middleware, to the best of our knowledge, that can concurrently schedule entities from multiple, different SPEs. Such scheduling can be useful when analysts deploy their queries in shared, central CPS devices, using multiple SPEs e.g., due to specific performance or compatibility requirements. To evaluate this scenario, we deploy VS, LR, and SYN on Storm, Flink, and Liebre in the Xeon server (23 queries total). The queries receive their inputs from separate Data Sources and at different rates, at a certain percentage of the empirically determined maximum rate each query can sustain in this setup. Lachesis enforces a multi-dimensional schedule, where each query is assigned to a cgroup, and given an equal part of the total CPU, using cpu.shares, whereas each operator is scheduled using the QS policy and nice. Figure 18 illustrates...
the performance of the OS and Lachesis with the QS policy. As shown, the benefits of Lachesis are consistent with the previous experiments on Odroids. All queries perform significantly better with Lachesis, which outperforms OS by up to 40% in throughput in the case of Liebre - SYN (60% of max rate for the OS 100% for Lachesis), up to 498x lower latency and 414x lower end-to-end latency in the case of Storm - VS (at input rate 100%).

6.7 Evaluation Summary
Our evaluation shows that Lachesis achieves the goals from §3.1, being able to use arbitrary scheduling policies to schedule one or more queries running possibly in multiple SPEs and nodes in both low- and high-end devices. Compared to OS, Lachesis helped queries attain significant performance improvements (up to 75% higher throughput, three orders of magnitude lower average latency, as well as two orders of magnitude lower average end-to-end latency and 99.9th percentile latency) while it also performed better than state-of-the-art UL-SS in a variety of experimental setups.

In all our experiments, the CPU utilization of Lachesis was very low (in most cases, around 1% of the total CPU and always less than 5%). Lachesis resulted in a slight increase in the CPU usage of Graphite (1-10%) due to the increased requests, but this did not negatively affect query performance.

7 RELATED WORK
As mentioned in §2, operator scheduling is sometimes used as a synonym for operator placement [2, 10, 34, 57–59]. Noting again that these are complementary techniques, we now further cover works about operator scheduling in line with Lachesis.

Scheduling in one-at-a-time SPEs. Haren [43] is a customizable UL-SS based on Liebre [52], which suffers from the drawbacks outlined in §1 and, in contrast to Lachesis, cannot be used with production-ready systems without significant changes. EdgeWise adopts a similar model for Storm [3, 18]. In contrast to Lachesis, it has a fixed policy (QS) and is only evaluated in single-query experiments. Being a UL-SS, EdgeWise cannot be added to different SPEs (or different versions of Storm) without changes to the SPE.

Lachesis applies custom scheduling through Linux mechanisms such as nice and cgroup [23]. cgroups are used in Storm [3] for coarse-grained control of resource allocation, and in resource controllers [27, 28], to adjust the computational capacity of each application in a shared platform and reduce QoS violations. In contrast to our work, such techniques are coarse-grained, cannot schedule individual operators nor support user-defined scheduling policies. Docker [29] uses cgroups to enforce resource limits for its containers. Lachesis can be adjusted to schedule queries deployed in docker by identifying the relevant cgroups and changing the scheduling attributes of operator threads according to the policy (e.g., updating their nice or altering relevant cgroup attributes).

Researchers have proposed diverse operator scheduling policies that Lachesis can support for arbitrary SPEs, i.e., Queue-Size (QS) [18], Highest-Rate (HR) [50], and FCFS [7] (cf. §6). Other existing policies are superseded by the above, i.e., the Rate-Based (RB) [55] policy, which minimizes the average latency of a single query (while the HR does the same for multiple queries [48]). The Chain policy [6] minimizes the memory usage of multiple queries and can take maximum query latency into account [5]. Other interesting policies optimize average query throughput (Min-Cost), average latency (Min-Latency, similar to HR and RB), available memory (Min-Memory) or the total QoS of the system [11, 12].

In executions of multiple heterogeneous queries, fairness can be important, i.e., minimizing the variation in query slowdown, measured by the slowdown or stretch [1, 39] metrics. Translating fairness-based policies for Lachesis can be an interesting future direction. The Longest Stretch First [1] policy minimizes the maximum slowdown, while other policies [49, 50] balance fairness and overall latency. Multi-class scheduling approaches [38] assign different QoS requirements to queries and also explore how scheduling and load management can work in synergy to honor priority classes [46, 47].

Scheduling for microbatched SPEs. Microbatched SPEs (cf. §1) are specifically optimized for throughput, and recent prototypes following this model have tried to exploit at best the computing power of single scale-up machines. For them, scheduling is referred to the logic behind the dynamic assignment of tasks to a pool of threads, where each task executes a chain of operators over the inputs of a batch (often in the order of thousands to amortize the scheduling overhead). Systems of this kind are StreamBox [36], based on a centralized scheduler, and LightSaber and Grizzly [21, 54]. The latter adopt a code generation approach where the task code is a tight loop generated and compiled from a SQL-like representation of the query, while tasks are scheduled using concurrent lock-free queues. Scheduling in those systems has a different goal than the one of one-at-a-time SPEs, often designed to balance the load among the threads in the pool rather than optimizing application-specific QoS requirements (e.g., average or maximum latency).

8 CONCLUSIONS
We presented Lachesis, a middleware for streaming applications that can enforce custom scheduling policies on streaming queries running in one or multiple nodes, using one or more SPEs. In contrast to previously proposed custom scheduling solutions, which required tight integration into the SPEs to schedule operators as user-level threads, Lachesis runs as a standalone process, using OS mechanisms such as nice and cgroup to guide the decisions of the OS scheduler, without altering SPEs or query implementations, or requiring query redeployment. Lachesis’ design is modular and can be extended to support new SPEs, policies, and OS mechanisms. We extensively evaluated Lachesis and showed that it can significantly outperform both the default OS scheduling adopted by SPEs as well as the SoA. Future work directions can include (1) the exploration of additional OS mechanisms, such as real-time threads and CPU quotas (available at Lachesis’ repository [41]), (2) global coordination for distributed Lachesis instances, (3) the usage of learning techniques to guide Lachesis’ scheduling, and (4) further exploration of query bottlenecks using pressure stall information [31].

ACKNOWLEDGMENTS
We thank the shepherd, Aniruddha Gokhale, and the anonymous reviewers. Work supported by the Swedish Government Agency for Innovation Systems VINNOVA, project “AutoSPADA” grant nr. DNR 2019-05884 in the funding program FFV Strategic Vehicle Research and Innovation, the Swedish Foundation for Strategic Research,


