EdgeWise: A Better Stream Processing Engine for the Edge

Xinwei Fu, *Virginia Tech*
Talha Ghaffar, *Virginia Tech*
James C. Davis, *Virginia Tech*
Dongyoon Lee, *Virginia Tech*

Abstract

Many Internet of Things (IoT) applications would benefit if streams of data could be analyzed rapidly at the Edge, near the data source. However, existing Stream Processing Engines (SPEs) are unsuited for the Edge because their designs assume Cloud-class resources and relatively generous throughput and latency constraints.

This paper presents EdgeWise, a new Edge-friendly SPE, and shows analytically and empirically that EdgeWise improves both throughput and latency. The key idea of EdgeWise is to incorporate a congestion-aware scheduler and a fixed-size worker pool into an SPE. Though this idea has been explored in the past, we are the first to apply it to modern SPEs and we provide a new queue-theoretic analysis to support it. In our single-node and distributed experiments we compare EdgeWise to the state-of-the-art Storm system. We report up to a 3x improvement in throughput while keeping latency low.

1 Introduction

Internet of Things (IoT) applications are growing rapidly in a wide range of domains, including smart cities, healthcare, and manufacturing [33, 43]. Broadly speaking, IoT systems consist of Things, Gateways, and the Cloud. Things are sensors that "read" from the world and actuators that "write" to it, and Gateways orchestrate Things and bridge them with the Cloud.

At the moment, IoT systems rely on the Cloud to process sensor data and trigger actuators. In principle, however, Things and Gateways could perform some or all data analysis themselves, moving the frontier of computation and services from the network core, the Cloud [17], to its Edge [21, 67], where the Things and Gateways reside. In this paper we explore the implications of this paradigm in a promising use case: *stream processing*.

Stream processing is well suited to the IoT Edge computing setting. Things generate continuous streams of data that often must be processed in a timely fashion; stream processing performs analysis on individual data points (*tuples*) rather than batches [24, 69]. As shown in Figure 1, stream processing is described by a directed acyclic graph, called a *topology*, whose vertices are data processing *operations* and edges indicate data flow.

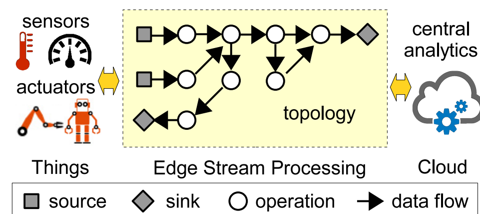

Figure 1: The Edge connects Things to the Cloud, and can perform local data stream processing.

Modern Stream Processing Engines (SPEs) such as Storm [15], Flink [13], and Heron [49] have mostly been designed for the Cloud, assuming powerful computing resources and plenty of memory. However, these assumptions do not hold at the Edge. In particular, these SPEs use a simple One Worker Per Operation Architecture (OWPOA). Given a dataflow topology and the degree of parallelism of each operation, OWPOA-style SPEs assign a dedicated *worker* thread to each *operation* instance, and link the worker-operation pairs with queues. Then, they rely on the operating system (OS) scheduler to choose which worker (operation) to schedule next, leading to *lost scheduling opportunities*: data propagates haphazardly through the topology because the scheduling of worker threads is left to the congestion-oblivious OS. With Cloud-scale resources these inefficiencies are amortized, but at the Edge they cannot be.

The database community studied efficient operation scheduling about a decade before modern SPEs came into vogue [18, 25]. Sadly, however, lessons learned from this

early research were not carried into the design of modern SPEs, leading to a sub-optimal performance when the existing OWPOA-style SPEs are used at the Edge setting. Existing IoT computing literature (e.g. [58, 64]) has ignored this history and has been building Edge computing platforms based on unmodified modern SPEs. This paper explores the impact of applying to modern SPEs these "lost lessons" of operation scheduling.

We present EdgeWise, an Edge-friendly SPE that revives the notion of engine-level operation scheduling to optimize data flows in a multiplexed (more operations than processor cores) and memory-constrained Edge environment. EdgeWise re-architects SPE runtime design and introduces an engine-level scheduler with a fixed-size worker pool where existing *profiling-guided* scheduling algorithms [18, 25] may be used. EdgeWise also proposes a *queue-length-based* congestion-aware scheduler that does not require profiling yet achieves equivalent (or better) performance improvement. EdgeWise monitors the numbers of pending data in queues, and its *scheduler* determines the highest-priority operation to process next, optimizing the flow of data through the topology. In addition, the EdgeWise's worker pool avoids unnecessary threading overheads and decouples the data plane (operations) from the control plane (workers).

We show analytically and empirically that EdgeWise outperforms Apache Storm [15], the exemplar of modern SPEs, on both throughput and latency. EdgeWise is a reminder of both the end-to-end design principle [65] and the benefits of applying old lessons in new contexts [61].

This paper provides the following contributions:

- We study the software architecture of existing SPEs and discuss their limitations in the Edge setting (§3). To the best of our knowledge, this paper is the first to observe the lack of operation scheduling in modern SPEs, a forgotten lesson from old SPE literature.
- We present EdgeWise, a new Edge-friendly SPE. EdgeWise learns from past lessons to apply an engine-level scheduler, choosing operations to optimize data flows and leveraging a fixed-size worker pool to minimize thread contention (§4).
- Using queuing theory, we argue analytically that our congestion-aware scheduler will improve both throughput and latency (§5). To our knowledge, we are the first to mathematically show balancing the queue sizes lead to improved performance in stream processing.
- We demonstrate EdgeWise's throughput and latency gains on IoT stream benchmarks (§7).

2 Background: Stream Processing

This section provides background on the stream processing programming model (§2.1) and two software architectures used in existing SPEs (§2.2).

2.1 Dataflow Programming Model

Stream processing uses the dataflow programming model depicted in the center of Figure 1 [24, 69]. Data *tuples* flow through a directed acyclic graph (*topology*) from *sources* to *sinks*. Each inner node is an *operation* that performs arbitrary computation on the data, ranging from simple filtering to complex operations like ML-based classification algorithms. In the Edge context a source might be an IoT sensor, while a sink might be an IoT actuator or a message queue to a Cloud service.

Though an operation can be arbitrary, the preferred idiom is *outer I/O, inner compute*. In other words, I/O should be handled by source and sink nodes, and the inner operation nodes should perform only memory and CPU-intensive operations [3, 41]. This idiom is based on the premise that the more unpredictable costs of I/O will complicate the scheduling and balancing of operations.

After defining the topology and the operations, data engineers convert the *logical* topology into a *physical* topology that describes the number of physical instances of each logical operation. In a distributed setting engineers can also indicate preferred mappings from operations to specific compute nodes. An SPE then deploys the operations onto the compute node(s), instantiates queues and workers, and manages the flow of tuples from one operation to another.

2.2 Stream Processing Engines

Starting from the notion of active databases [55, 76], early-generation SPEs were proposed and designed by the database community in the early 2000s: e.g. Aurora [24], TelegraphCQ [27], Stream [16], and Borealis [7, 28]. Interest in SPEs led to work on performance optimization techniques such as operation scheduling [18, 25] and load shedding [35, 70]. We will revisit these works shortly (§3.4).

The second generation of "modern SPEs" began with Apache Storm [15] (2012) as part of the democratization of big data. Together with Apache Flink [13] and Twitter's Heron [49], these second-generation SPEs have been mainly developed by practitioners with a focus on scalable Cloud computing. They have achieved broad adoption in industry.

Under the hood, these modern SPEs are based on the One Worker Per Operation Architecture (OWPOA, Figure 2). In the OWPOA, the operations are connected by queues in a pipelined manner, and processed by its

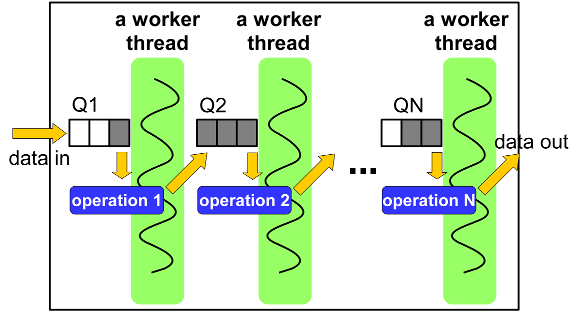

Figure 2: One Worker Per Operation Architecture (OWPOA). It assigns a worker thread for each operation. The example shows the case with N operations. Q represents a queue where a gray box means a pending data.

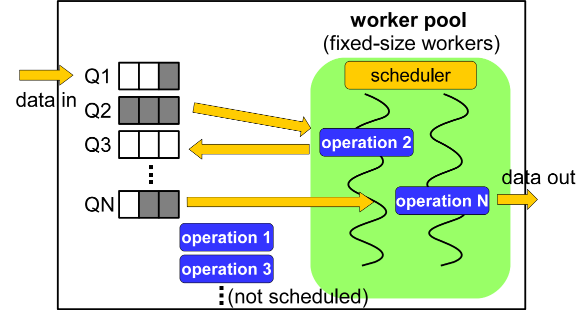

Figure 3: EDGEWISE Architecture (§4). The scheduler picks which operation to run. In this example, operation 2 and operation N had the longest queues, so they were scheduled.

own workers. Some operations may be mapped onto different nodes for distributed computing or to take advantage of heterogeneous resources (GPU, FPGA, etc.). In addition, the OWPOA monitors the health of the topology by checking the lengths of the per-operation queues. Queue lengths are bounded by a *backpressure* mechanism [32, 49], during which the source(s) buffer input until the operation queues clear.

3 Edge SPE Requirements Analysis

Stream processing is an important use case for Edge computing, but as we will show, existing SPEs are unsuited for the Edge. This section discusses our Edge SPE requirements analysis and the drawbacks of existing SPEs.

3.1 Our Edge Model

We first present our model for the Edge.

Hardware. Like existing IoT frameworks (e.g. Kura [36], EdgeX [2], and OpenFog [5]), we view the Edge as a distributed collection of IoT Gateways that connect Things to the Cloud. We consider Edge stream processing on IoT Gateways that are reasonably stable and well connected, unlike mobile drones or vehicles. We further assume these IoT Gateways have limited computing resources compared to the Cloud: few-core processors, little memory, and little permanent storage [17, 67]. However, they have more resources than those available to embedded, wireless sensor networks [50], and thus can afford reasonably complex software like SPEs. For example, Cisco's IoT Gateways come with a quad-core processor and 1GB of RAM [31].

Applications. Edge topologies consume IoT sensor data and apply a sequence of reasonably complex operations: e.g. SenML [44] parsers, Kalman filters, linear regressions, and decision tree classifications. For example, FarmBeats [72], a smart farm platform, uses IoT Gateways to collect data from various sensors and drones, to create summaries before sending them to the Cloud for long-term and cross-farm analytics, and to perform time-sensitive local data processing. We therefore assume a diverse set of workloads ranging from simple extraction-transform-load (ETL) and statistical summarization to predictive model training and classification.

3.2 Edge SPE Requirements

Every SPE aims for high throughput, and Edge SPEs are no exception. In addition, we see the following unique requirements for the Edge:

(1) Multiplexed. An Edge SPE must support an arbitrary topology on limited resources. We particularly focus on supporting a single topology in which there are more operations than processors, such that operation executions must be multiplexed on limited processors. Where a server-class machine can use threads unconcernedly, an Edge-class machine must be wary of unnecessary overheads.

(2) Low latency. An Edge SPE must offer low latency, else system architects should simply transmit raw data to the Cloud for analysis.

(3) No backpressure. The backpressure mechanism in Cloud SPEs is inadvisable at the Edge for two reasons. First, it is a "stop the world" scheme that destroys system latency, but latency is critical for Edge workloads. Second, it assumes that a data source can buffer pending data. While the Cloud can assume a persistent data source such as Kafka [4], at the Edge there is nowhere to put this data. Modern "real-time" SPEs such as Storm and Flink do not support file I/O based buffering. As a result the SPE must be congestion-aware to ensure queue lengths do not exceed available memory.

(4) Scalable. The SPE must permit scaling across multiprocessors within an IoT Gateway and across multiple Gateways, especially to take advantage of locality or heterogeneity favorable to one operation or another.

3.3 Shortcomings of OWPOA-style SPEs

The OWPOA naturally takes advantage of intra-node and inter-node parallelism, especially when data engineers make a good logical-to-physical mapping. In the OWPOA, however, *multiplexing* becomes challenging in complex topologies, because each logical operation must have at least one worker and there may be more workers than cores on a compute node. If there are too many workers assigned to one compute node the poor scheduling of workers will artificially limit performance.

Let us explain the issue in detail. The OWPOA relies on the OS scheduler to decide which worker-operation pair to schedule next. If the input rate is low enough that most queues are empty (i.e. the SPE is over-provisioned), there is no harm in such an approach. Only some operations will have work to do and be scheduled, while the remainder will sleep.

But if the input rate rises (equivalently, if the SPE becomes less provisioned) then the SPE will become *saturated*, i.e. most or all queues will contain tuples. In this case any operation might be scheduled by the OS. As some operations take longer than others, a typical round-robin OS scheduler will naturally imbalance the queue lengths and periodically trigger backpressure. For example, in Figure 2, although Queue 2 is full, the OS may unwisely schedule the other worker-operation pair first, triggering *backpressure* unnecessarily and leading to significantly high *latency*.

3.4 A Lost Lesson: Operation Scheduling

As noted earlier (§2.2), the database community has studied different *profiling-guided* priority-based operation scheduling algorithms [18, 25] in the context of multiple operations and a single worker, before the modern OWPOA-style SPEs were born. For instance, Carney et al. [25] proposed a "Min-Latency" algorithm which assigns higher (static) priority on latter operations than earlier operations in a topology and processes old tuples in the middle of a topology before newly arrived tuples, with a goal to minimize average latency. For a topology with multiple paths, the tie is broken by the profiled execution time and input-output ratio of each operation. With a goal to minimize queue memory sizes, Babcock et al. [18] proposed a "Min-Memory" algorithm (called Chain) which favors operations with higher input-output reduction and short execution time (e.g., faster filters).

Unfortunately, however, modern OWPOA-style SPEs (e.g. Storm [15], Flink [13], Heron [49]) have not adopted these research findings when designing multi-core multi-worker SPEs and simply relied on the congestion-oblivious OS scheduler. *This paper argues that Edge SPEs should be rearchitected to regain the benefits of engine-level operation scheduling to optimize data flows in a multiplexed Edge environment.* In particular, we compare the effectiveness of both profiling-based (old) and dynamic balancing (new) scheduling algorithms, and report that all offer significant throughput and latency improvements over modern SPEs.

4 Design of EdgeWise

This section presents EdgeWise, an Edge-friendly SPE, that leverages a congestion-aware scheduler (§4.1) and a fixed-size worker pool (§4.2), as illustrated in Figure 3. EdgeWise achieves higher throughput and low latency by *balancing the queue lengths*, with the effect of *pushing the backpressure point to a higher input rate*. Thus EdgeWise achieves higher throughput without degrading latency (no backpressure). We later analyze the improved performance mathematically in §5.

4.1 Congestion-Aware Scheduler

EdgeWise addresses the scheduling inefficiency of the OWPOA by incorporating a user-level scheduler to make wiser choices. In the OWPOA design, physical operations are coupled to worker threads and are scheduled according to the OS scheduler policy. The EdgeWise scheduler separates the threads (execution) from the operations (data). Prior work has proposed using profiling-based operation scheduling algorithms [18, 25]. Instead, we propose a profiling-free dynamic approach that balances queue sizes by assigning a ready thread to the operation with the most pending data.

The intuition behind EdgeWise is shown in Figure 4, which compares the behavior of an OWPOA-style SPE to EdgeWise in a multiplexed environment. Figure 4(a) shows the unwise choice that may be made by the random scheduler of the OWPOA, leading to backpressure (high latency). Figure 4(b) contrasts this with the choice made by EdgeWise's congestion-aware scheduler, evening out the queue lengths to avoid backpressure.

We believe EdgeWise's congestion-aware scheduler would be beneficial to the Cloud context, as an intra-node optimization. In practice, however, we expect that EdgeWise will have greater impact in the Edge setting, where (1) with few cores, there are likely more operators than cores, (2) latency is as critical as throughput, and (3) memory is limited.

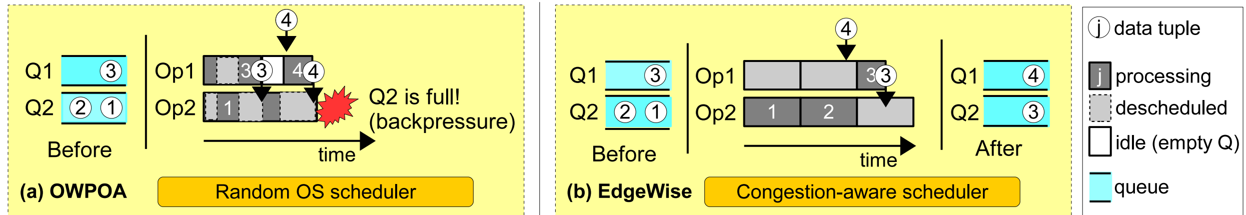

Figure 4: A scheduling example with two operations, multiplexed on a single core. The Q(ueue) size is two. Initially, Q1 has one tuple and Q2 is full. (a) The random OS scheduler in OWPOA lets Op(eration) 1 process tuples ③ and (newly coming) ④ first, overflowing Q2 and triggering unnecessary backpressure. (b) EDGEWISE knows Q2 has more pending data, and thus schedules the more critical Op2 first, avoiding congestion.

4.2 Fixed-size Worker Pool

EDGEWISE's scheduler decouples data from execution, requiring some changes in how EDGEWISE realizes the physical topology supplied by data engineers. Rather than dedicating a worker to each operation as in the OWPOA, EDGEWISE processes data on a fixed set of workers in the worker pool (Figure 3). These workers move from operation to operation as assigned by the scheduler. EDGEWISE could in principle work solely from the logical topology, in effect dynamically rearranging the physical topology (# workers assigned to each operation). However, this would violate any assumptions about thread safety or physical topology embedded in the logical topology. Thus, EDGEWISE uses the existing physical topology to bound the number of workers assigned to any operation at one time.

The worker pool size is configurable. By default it is set to the number of processors, because we assume that most operations are CPU-intensive per the recommended stream programming model (§2.1). If there are many I/O-bound operations, a fixed-size worker pool may lead to sub-optimal performance, requiring users to tune the pool size.

Putting it together. The scheduler dynamically chooses which operation a worker should perform. When a worker is ready, it asks the scheduler for an assignment, and the scheduler directs it to the operation with the longest pending data queue. Without work, it sleeps. When new data arrives, the scheduler wakes up the worker. The worker *non-preemptively*[1] completes the operation, and EDGEWISE directs the output to the downstream operation queue(s). Different operations may run in parallel: e.g., operations `2` and `N` in Figure 3. However, for FIFO guarantees, EDGEWISE schedules at most one worker to any operation instance (i.e. to any queue).

EDGEWISE supports three different data consumption policies for each scheduling turn: (1) `All` consumes all the tuples in the queue; (2) `Half` consumes half of the tuples in the queue; and (3) `At-most-N` consumes at most `N` tuples in the queue. Intuitively, in a saturated system, consuming a small constant number in each quantum will cause the scheduler to make more decisions overall. As our scheduler is fast, the more decisions it makes the better it should approximate the ideal schedule (evaluated in §7.4). Such consumption policies are not possible in OWPOA-style SPEs.

EDGEWISE meets the Edge SPE design goals, listed in §3.2. EDGEWISE retains the achievements of the OWPOA, and to these it adds the multiplexed, low-latency, and no-backpressure requirements. EDGEWISE's scheduler allows it to balance queue lengths, improving average latency and avoiding the need for backpressure by keeping heavier operations from lagging behind.

5 Performance Analysis of EDGEWISE

This section shows analytically that EDGEWISE will achieve higher throughput (§5.1) and lower latency (§5.2) than OWPOA. Lastly, in §5.3 we discuss how to measure relevant runtime metrics for the performance analysis. To the best of our knowledge, we are the first to apply queueing theory to analyze the improved performance in the context of stream processing. Prior scheduling works in stream processing either provide no analysis [25] or focus only on memory optimization [18].

5.1 Higher Throughput

First we show that maximum end-to-end throughput depends on scheduling heavier operations proportionally more than lighter operations. This is impossible to guarantee in the scheduler-less modern SPEs, but easy for EDGEWISE's scheduler to accomplish.

Our analysis interprets a dataflow topology as a queuing network [38, 46]: a directed acyclic graph of stations. Widgets (tuples) enter the network via the queue of the first station. Once the widget reaches the front of a station's queue, a server (worker) operates on it, and then it advances to the next station. The queuing theory model

[1] More precisely, it is non-preemptive at the engine level. EDGEWISE can still be preempted by the underlying OS scheduler.

allows us to capture the essential differences between EDGEWISE and the OWPOA. In the rest of this section we draw heavily on [38, 46], and we discuss the model's deviations from real topology behavior at the end.

Modeling. Given the input rate λ_i and the service rate μ_i of a server i, the server utilization ρ_i (fraction of time it is busy) is defined as:

$$\rho_i = \frac{\lambda_i}{\mu_i} \tag{1}$$

A queuing network is *stable* when $\rho_i < 1$ for all servers i. If there is a station to which widgets arrive more quickly than they are serviced, the queue of that station will grow unbounded. In an SPE, unbounded queue growth triggers the backpressure mechanism, impacting system latency.

Suppose we want to model a topology with M operations in this way. We can represent the input and server rates of each operation as a function of λ_0 and μ_0, the input and service rates of the first operation, respectively. We are particularly interested in λ_0 as it is the input rate to the system; higher λ_0 means higher throughput. Expressing the input scaling factors and relative operation costs for an operation i as q_i and r_i, we can write the input rate λ_i and the service rate μ_i for operation i as:

$$\lambda_i = q_i \cdot \lambda_0 \qquad \mu_i = r_i \cdot \mu_0$$

In queuing theory, each node is assumed to have an exclusive server (worker). However, in the Edge, there may be fewer servers (processor cores) than stations (operations). If there are C processor cores, we can model processor contention by introducing a scheduling weight w_i, subject to $\sum_i^M w_i = C$, yielding the effective service rate $\mu_i{}'$:

$$\mu_i{}' = w_i \cdot \mu_i = w_i \cdot (r_i \cdot \mu_0)$$

For instance, when one core is shared by two operations, a fair scheduler halves the service rate ($w_1 = w_2 = \frac{1}{2}$) as the execution time doubles. The constraint $\sum_i^M w_i = C$ reflects the fact that C processor cores are shared by M operations.

Similarly, the effective server utilization $\rho_i{}'$ can be redefined based on $\mu_i{}'$. To keep the system stable (eliminate backpressure), we want $\rho_i{}'$ to be less than one:

$$\forall i, \quad \rho_i{}' = \frac{\lambda_i}{\mu_i{}'} = \frac{q_i \cdot \lambda_0}{w_i \cdot r_i \cdot \mu_0} < 1 \tag{2}$$

which can be rearranged as

$$\forall i, \quad \lambda_0 < w_i \cdot \frac{r_i}{q_i} \cdot \mu_0 \tag{3}$$

Optimizing. Remember, λ_0 represents the input rate to the system and is under our control; a higher input rate means higher throughput (e.g. the system can sample sensors at a higher rate). Assuming that μ_0, r_i, and q_i are constants for a given topology, the constraint of Equation (3) implies that the maximum input rate λ_0 that an SPE can achieve is upper-bounded by the minimum of $w_i \cdot \frac{r_i}{q_i}$ over all i. Thus, our optimization objective is to:

$$\begin{aligned} \text{maximize} \quad & \min_i (w_i \cdot \frac{r_i}{q_i}) \\ \text{subject to} \quad & \sum_i^M w_i = C \end{aligned} \tag{4}$$

It is straightforward to show that the maximum input rate can be achieved when $w_i \cdot \frac{r_i}{q_i}$ are equal to each other for all i: i.e.,

$$w_1 : w_2 : \cdots : w_N = \frac{q_1}{r_1} : \frac{q_2}{r_2} : \cdots : \frac{q_M}{r_M}$$

In other words, the scheduling weight w_i should be assigned proportional to $\frac{q_i}{r_i}$. This is indeed very intuitive. An operation becomes heavy-loaded when it has a higher input rate (higher q_i) and/or a lower service rate (lower r_i). Such an operation should get more turns.

Scheduling. EDGEWISE's scheduler dynamically identifies heavily loaded operations and gives them more turns. The queue of an operation with higher input rate and higher compute time (lower service rate) grows faster than the others. By monitoring queue sizes, we can identify and favor the heavy-loaded operations with more frequent worker assignments.

Compare this behavior to that of the OWPOA. There, the "fair" OS scheduler blindly ensures that w_i is the same for each operation, unaware of relative input and service rates, leading to suboptimal throughput. Once saturated, one of the heavy operations will reach the utilization cap of $\rho_i = 1$, become a bottleneck in the topology, and eventually trigger backpressure and latency collapse. Data engineers can attempt to account for this by profiling their topologies and specifying more workers for heavier operations. Fundamentally, however, this is an ad hoc solution: the OS scheduler remains blind.

In our evaluation (§7.3.1) we show that EDGEWISE achieves $w_i \cdot \frac{r_i}{q_i}$ equality across operations at runtime, leading to an optimal balanced effective server utilization $\rho_i{}'$. In contrast, we report that in the OWPOA approach, increasing the input rate leads to increasingly unbalanced $\rho_i{}'$ across operations.

Runtime deviations from the model. Our queuing theory model captures real SPE behavior in the essentials, but it deviates somewhat in the particulars. We give two prominent examples. The first deviation is that our queuing theory model assumes predictable widget fan-in and fan-out ratios at each operator (i.e. constant q_i and r_i). In reality these are distributions. For example, an operation that splits a sentence into its constituent words

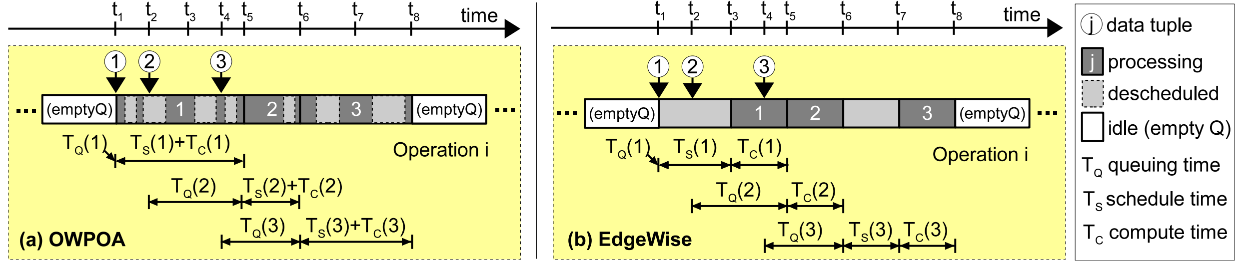

Figure 5: A per-operation latency breakdown example of (a) OWPOA, and (b) EDGEWISE. Incoming data tuples ①, ②, and ③ are being queued. When scheduled (dark gray box), the operation i processes a pending tuple. A tuple j can be in one of three states: waiting in the middle of the queue (T_Q), at the head of the queue waiting to be scheduled (T_S), or being computed (T_C). Later in §7.3.2, we show that T_Q dominates the per-operation latency in a saturated system.

will emit one tuple for "Please", two tuples for "Please accept", and so on. The second deviation is that we assumed that operations have constant costs. In reality it will take such a sentence-splitting operation more time to process longer sentences.

Considering these deviations, the difficulty data engineers face when generating physical topologies for a OWPOA SPE is clear: it is difficult to identify a single integer quantity of workers to assign to each operation, and even then balance will only be achieved probabilistically over many scheduling cycles. In contrast, EDGEWISE automatically balances the assignment of workers to operations.

5.2 Lower Latency

Now we show that EDGEWISE can achieve lower end-to-end *latency* than the OWPOA without compromising throughput. Our analysis hinges on the observation that unbalanced queue lengths have an outsized effect on latency, so that balancing queue lengths leads to an overall improvement in latency.

The total latency for a completed tuple equals the sum of the latencies paid at each operation it visited on its way from source to sink, plus communication latencies outside of our control:

$$Latency = \sum_{i}^{M}(L_i + Comm.) \approx \sum_{i}^{M} L_i \qquad (5)$$

In stream processing, the per-operation latency consists of (1) the queue time T_Q, waiting in the queue; (2) the schedule time T_S, waiting for a turn at the head of the queue; and (3) the (pure) compute time T_C, being processed. As illustrated in Figure 5, these times come at a different form. In OWPOA (a), the preemptive OS scheduler makes T_S and T_C interleaved. In EDGEWISE (b), the non-preemptive scheduler makes clear distinction because an operation is not scheduled until a worker is available, and when scheduled, it completes its work.

In a saturated system, we treat T_S and T_C as constants, while T_Q will grow with the input rate to dominate L_i.

$$L_i = T_Q + (T_S + T_C) \approx T_Q \qquad (6)$$

Assuming that the input and service rates, λ and μ, can be modeled as exponential random variables[2], Gross et al. [38] show that the queue time T_Q can be expressed as

$$T_Q = \frac{\rho}{\mu - \lambda} = \frac{\lambda}{\mu(\mu - \lambda)} \qquad (7)$$

Note that T_Q has a vertical asymptote (approaches ∞) at $\mu = \lambda$, and a horizontal asymptote (approaches 0) when $\mu >> \lambda$. In other words, for a given λ, a tuple will wait longer in the queues of heavier operations, and crucially *the growth in wait time is non-linear (accelerates) as μ approaches λ*. This means that heavier operations have a much larger T_Q and thus L_i (Equation (6)) than lighter operations, and an outsized impact on overall latency (Equation (5)).

Though the effect of heavy operations may be dire when using the OWPOA's random scheduler, EDGEWISE's congestion-aware scheduler can avoid this problem by giving more hardware resources (CPU turns) to heavier operations over lighter ones. In effect, EDGEWISE balances the T_Q of different operations, reducing the T_Q of tuples waiting at heavy operations but increasing the T_Q of tuples waiting at lighter operations. According to Gross et al.'s model this balancing act is not a zero-sum game: we expect that the lighter operations sit near the horizontal asymptote while the heavier operations sit near the vertical asymptote, and balancing queue times will shift the entire latency curve towards the horizontal asymptote. In our evaluation we support this analysis empirically (§7.3.2).

[2]For λ and μ with general distributions, the curve in the $\lambda < \mu$ region is similar. The exponential model simplifies the presentation.

5.3 Measuring Operation Utilization

In our evaluation we compare EDGEWISE with a state-of-the-art OWPOA. For a fine-grained comparison, in addition to throughput and latency we must compare the operation utilization $\rho = \frac{\lambda}{\mu}$ discussed in §5.1. This section describes how we can fairly compare EDGEWISE against a baseline OWPOA system in this regard.

We want to compare the server (operation) utilization of EDGEWISE and the baseline. As argued in §5.1, an ideal result is a balanced utilization vector consisting of equal values $1-\varepsilon$, with ε chosen based on latency requirements. As operation utilization may change over the lifetime of an SPE run (e.g. as it reaches "steady-state"), we incorporate a time window T_w into the metric given in Equation (1).

During a time window T_w, an operation might be performed on N tuples requiring a total of T_E CPU time. The input rate for this operation during this window is $\lambda_w = \frac{N}{T_w}$ and the service rate is $\mu_w = \frac{1}{T_E}$, so we have windowed utilization ρ_w as:

$$\rho_w = \frac{\lambda_w}{\mu_w} = T_E \cdot \frac{N}{T_w} \quad (8)$$

Unsurprisingly, we found that this is the utilization metric built into Storm [15][3], the baseline system in our evaluation.

In the OWPOA, computing ρ_w is easy. T_w is fixed, N is readily obtained, and T_E can be calculated by monitoring the beginning and ending time of performing the operation on each tuple:

$$T_E = \frac{1}{N}\sum_{j}^{N}(T_{end}(j) - T_{begin}(j)) \quad (9)$$

Equation (9) would suffice to measure ρ_w for the OWPOA, but for a fair comparison between the OWPOA and EDGEWISE we need a different definition. Note that $T_{end}(j) - T_{begin}(j)$ captures the total time a worker spends applying an operation to a tuple, and in the OWPOA (Figure 5 (a)) this calculation includes both T_C (pure computation) and T_S (worker contention). As EDGEWISE's scheduler is non-preemptive (Figure 5 (b)), measuring T_E in this way would capture only EDGEWISE's T_C. Doing so would ignore its T_S, the time during which EDGEWISE's scheduler decides *not* to schedule an operation with a non-empty queue in favor of another operation with a longer queue. This would artificially decrease T_E and thus ρ_w for this operation.

To capture T_S for an operation in EDGEWISE, we can instead amortize the schedule time T_S across all completed tuples, and describe the operation's time during this window as spent either executing or idling with an empty queue:

$$T_w = T_E \cdot N + T_{emptyQ} \quad (10)$$

Solving Equation (10) for T_E and substituting into Equation (8), we can compute ρ_w in EDGEWISE:

$$\rho_w = 1 - \frac{T_{emptyQ}}{T_w} \quad (11)$$

The windowed utilization metric ρ_w given in Equation (11) applies equally well to SPEs that use the OWPOA or EDGEWISE.

[3] In Apache Storm this metric is called the operation's *capacity*.

6 Implementation

We implemented EDGEWISE on top of Apache Storm [15] (v1.1.0). Among modern SPEs, Storm was the most popular for data science in 2018 [20] and has the lowest overall latency [29]. We made three major modifications: (1) We implemented the congestion-aware scheduler (§4.1), and added two data structures: a list of operations with non-empty queues as scheduling candidates; and a list of running operations to ensure FIFO per queue; (2) We removed the per-operation worker threads, and added one worker pool of (configurable) K worker threads (§4.2); and (3) We introduced two queuing-related metrics, T_{emptyQ} and T_Q, for server utilization and latency breakdown analysis (§5.3).

EDGEWISE can be applied to other OWPOA-style SPEs such as Flink [13] and Heron [49]. In Flink, multiple operators may be grouped in a single Task Slot, but each operator (called "subtask") still has its own worker thread. Task Slot separates only the managed "memory" of tasks, but there is no CPU isolation. As a result, worker threads will still contend and cause congestion if there are more operators than CPUs. Thus, EDGEWISE's congestion-aware scheduler and fixed-size worker pool will be equally beneficial for Flink.

The EDGEWISE prototype is available at `https://github.com/VTLeeLab/EdgeWise-ATC-19`. It adds 1500 lines of Java and Clojure across 30 files.

7 Evaluation

Our evaluation first shows EDGEWISE's throughput-latency performance on representative Edge stream processing workloads (§7.2), followed by detailed performance analysis (§7.3). Then we present a sensitivity study on different consumption policies (§7.4), and a distributed (inter-node) performance study (§7.5).

7.1 General Methodology

Hardware. EDGEWISE is designed for the Edge, so experiments use an intermediate-class computing device

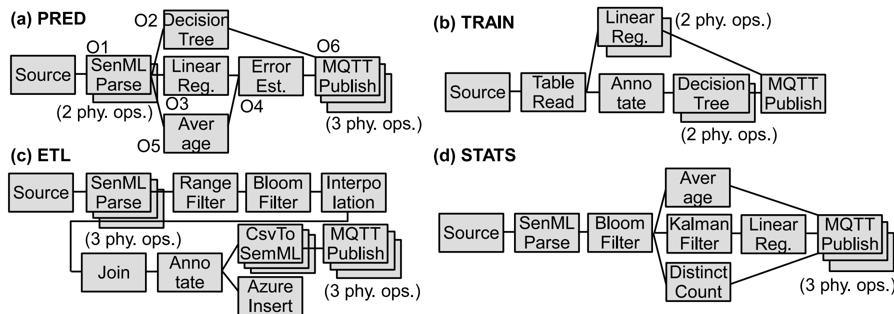

Figure 6: Tested Topologies in RIoTBench [68]: (a) PREDictive analytics, (b) model TRAINinig, (c) Extract, Transform, and Load, and (d) STATistical summarization. The (tuned) number of physical operations are shown as the overlapped, multiple rectangles: e.g., 3 MQTT Publish operations in (a).

representative of IoT Edge devices (§3.1): weaker than Cloud-class servers but stronger than typical embedded systems. Specifically, we use Raspberry Pi 3 Model B devices (raspbis), which have a 1.2GHz quad-core ARM Cortex-A53 with 1GB of RAM and run Raspbian GNU/Linux 8.0 v4.1.18.

Baseline. We used Storm [15] as the OWPOA baseline. We set up Storm's dependencies, Nimbus and a Zookeeper server, on a desktop machine, and placed the Storm supervisors (compute nodes) on the raspbis. Because EDGEWISE optimizes the performance of a single compute node, all but our distributed experiment (§7.5) use a single raspbi. As our raspbis are quad-core, EDGEWISE uses four worker threads.

Schedulers. In addition to EDGEWISE's queue-length-based approach, we also evaluated implementations of the Min-Memory [18] and Min-Latency [25] schedulers, as well as a Random scheduler. All schedulers used our `At-most-50` data consumption policy (§4.2), based on our sensitivity study §7.4.

Benchmarks. We used the RIoTBench benchmark suite [68], a real-time IoT stream processing benchmark implemented for Storm[4]. The RIoTBench benchmarks perform various analyses on a real-world Smart Cities data stream [22]. Each input tuple is 380 bytes. The sizes of intermediate tuples vary along the topology.

Figure 6 shows RIoTBench's four topologies. We identified each physical topology using a search guided by the server utilization metric (§5.1), resulting in physical topologies with optimal latency-throughput curves on our raspbi nodes [40]. We used the same physical topologies for both Storm and EDGEWISE.

We made three modifications to the RIoTBench benchmarks: (1) We patched various bugs and inefficiencies. (2) We replaced any Cloud-based services with lab-based ones; (3) To enable a controlled experiment, we implemented a timer-based input generator that reads data from a replayed trace at a configurable input rate. To be more specific, the Smart Cities data [22] is first loaded into the memory, and a timer periodically (every 100 ms) feeds a fixed-size batch of them into `Spout`, the source operator in Storm. We changed the batch size to vary the input rate. This simulates a topology measuring sensor data at different frequencies, or measuring different number of sensors at a fixed frequency.

Metrics. Measurements were taken during the one minute of steady-state runs, after discarding couple minutes of initial phase. We measured *throughput* by counting the number of tuples that reach the `MQTT Publish` sink. Measuring throughput at the sink results in different throughput rates for each topology at a given input rate, since different topologies have input-output tuple ratios: e.g., 1:2 in `PRED`, 1:5 in `STATS`. We measured *latency* by sampling 5% of the tuples, assigning each tuple a unique ID and comparing timestamps at source and the same sink used for the throughput measurement. We measured *operation utilization* using ρ_w (Equation (11)). Each experiment was performed 5 times with each configuration. The error bars indicate one standard deviation from the average. Most data points had small variances.

7.2 Throughput-Latency Performance

We measured the throughput-latency performance curve for each of the RIoTBench applications on one raspbi across a range of input rates. The curves for `PRED`, `STATS`, and `ETL` are shown in Figure 7; the curve for the `TRAIN` application (not shown) looks like that of the `ETL` application. In general, both Storm and EDGEWISE have excellent performance when the system is under-utilized (low throughput). Latency performance collapses at high throughput rates as a result of frequent backpressure.

The results show that an SPE with an engine-level operation scheduler and a worker pool (WP) significantly outperforms Storm (an OWPOA-based SPE) at the Edge, in effect shifting the Storm throughput-latency curve down (lower latency) and to the right (higher throughput). First, the gaps between Storm and WP+Random indicate

[4] Other benchmarks [6, 53] seemed too unrealistic for our use case.

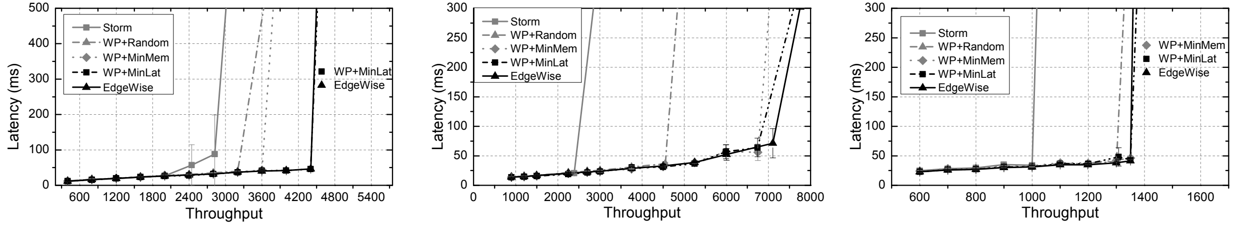

Figure 7: Throughput-latency of (a) PRED, (b) STATS, and (c) ETL topologies.

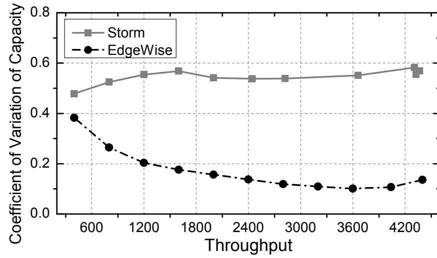

Figure 8: In PRED, as the input rate (thus throughput) increases, the coefficient of variation (CV) of capacities grows in Storm, but it decreases in EDGEWISE.

the benefit of avoiding unnecessary contentions in OWPOA. More importantly, the engine-level schedulers in effect push the backpressure point to a higher input rate, allowing the SPEs to achieve higher throughput at a low latency. The high variance in the Storm curves at higher throughput rates indicate early, random backpressure.

Among the variants that use a scheduler and WP, EDGEWISE's queue-length-based scheduler matches or outperforms WP+MinLat on latency and throughput, while WP+MinMem leads to improved (but not the best) performance as it is optimized for memory. Note that EDGEWISE does not require profiling per-operation execution time and input-output ratio as do MinLat and MinMem. For PRED, while keeping the latency low (say $\leq$100 ms), EDGEWISE improves its throughput by 57% (from 2800 to 4400). EDGEWISE is particularly effective in the STATS, where it achieves a 3x throughput improvement with low latency. For ETL, EDGEWISE improves throughput from 1000 to 1350 under 50ms latency.

7.3 Detailed Performance Breakdown

This experiment investigates the underlying causes of the throughput and latency gains described in §7.2, lending empirical support to our analysis in §5. We use the PRED application as a case study, though results were similar in the other RIoTBench applications.

7.3.1 Fine-Grained Throughput Analysis

Our analysis of throughput in §5.1 predicted that balancing effective server (operation) utilization would yield gains in throughput. To measure the extent to which Storm and EDGEWISE balance server utilization, in this experiment we calculated the windowed utilization ρ_w of each operation using Equation (11) and then computed the coefficient of variation ($CV = \frac{stddev}{avg}$) of this vector. A lower utilization CV means more balanced server utilization.

Figure 8 plots the coefficient of variation for Storm and EDGEWISE for different input rates. As the input rate (and thus output throughput) increases, the utilization CV increases in Storm, indicating that the operations become unbalanced as Storm becomes saturated. In contrast, in EDGEWISE the utilization CV decreases for larger input rates (and the raw ρ_w values approach 1). As predicted, EDGEWISE's throughput gains are explained by its superior ρ_w balancing.

7.3.2 Fine-Grained Latency Analysis

Our analysis of latency in §5.2 noted that in a congestion-blind scheduler, heavier operations would develop longer queues, and that the queuing time at these operations would dominate end-to-end latency. We showed that an SPE that reduced queuing times at heavy operations, even at the cost of increased queuing times at lighter operations, would obtain an outsized improvement in latency. In this experiment we validate this analysis empirically.

For this experiment we break down per-operation latency into its constituent parts (Equation (6)) and analyze the results in light of our analysis.

Figure 9 shows the per-operation latency for the 6 non-source operations in the PRED topology in the baseline Storm. Its logical and physical topology is shown in Figure 6 (a). Where our physical topology has multiple instances of an operation (two instances of O1, three of O6), we show the average queuing latency across instances.

Note first that the queue time T_Q dominates the per-operation latency L_i. Then, as the input rate increases from L(ow) to H(igh), the queues of heavier operations (O1, O6) grow much longer than those of lighter operations, and the queue time at these operations dominates the overall latency. Also note that the latency of lighter operations may decrease at high throughput, as tuples are

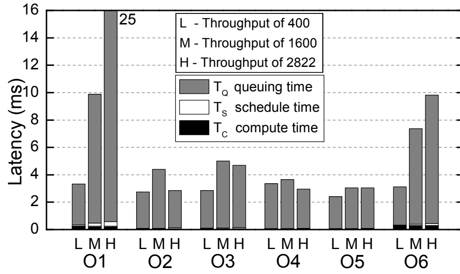

Figure 9: In Storm, as the throughput increases from L(ow) to M(edium) to H(igh), the queuing latency of heavy operations (e.g., `O1` and `O6`) increases rapidly.

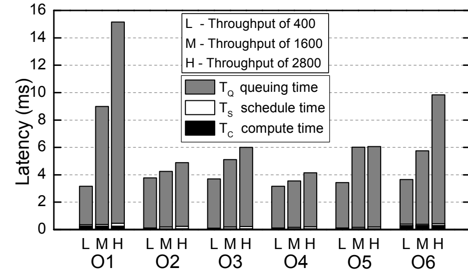

Figure 10: In EDGEWISE, as the throughput increases, the queuing latency of heavy operations (e.g., `O1` and `O6`) increases slowly.

mostly waiting in the queue of heavy operations, reflecting the effects of the non-deterministic OS scheduler.

In contrast, we observed different behavior for EDGEWISE as shown in Figure 10. The heaviest operation `O1` is still noticeable but its queue time under High input rate is only 15 ms, much smaller than the 25 ms of Storm. Of course this penalizes the lighter operations, but as we argued in §5.2 this is not a zero-sum game; the improvement in end-to-end latency outweighs any small per-operation latency increase.

The schedule time T_S includes both the scheduling overhead and the waiting time due to contention. Across all throughput rates and operations, T_S remains very small, implying that the overhead of EDGEWISE's scheduler is negligible.

7.4 Data Consumption Policy

In this experiment we explored the sensitivity of EDGEWISE's performance to its data consumption policies: a constant number (`At-most-N`) or a number proportional to the queue length (`All`, `Half`).

In Figure 11 you can see the effect of these rules in the `STATS` topology, as well as the Storm performance for comparison. As expected, the constant consumption rules consistently performed well in `STATS`. The `PRED` showed the trend similar to the `STATS`. The `TRAIN` and `ETL` topologies were not sensitive to the consumption policy. The `At-most-50` rule (solid black line) offers good latency with the highest throughput for all, so we used it in our other experiments.

7.5 Performance on Distributed Edge

Streaming workloads can benefit from scaling to multiple compute nodes, and supporting scaling was one of our design goals (§3.2). In this experiment we show that EDGEWISE's intra-node optimizations benefit an inter-node (distributed) workload.

We deployed the `PRED` application across 2, 4, and 8 raspbis connected on a 1G Ethernet lab network, simply

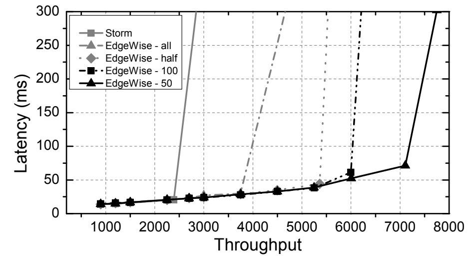

Figure 11: Sensitivity study on various consumption policies with `STATS` topology.

increasing the physical operation instances proportional to the number of raspbis and assigning them uniformly across nodes. Identifying an optimal distributed topology (and optimal partitioning across nodes) is out of scope for our work, which focuses on the optimal scheduling of the topology deployed on a single node. Experiments on visionary hundred- or thousand-node cases are left for future work. We used the same methodology as in §7.2 to collect metrics.

As expected, EDGEWISE's intra-node optimizations prove beneficial in an inter-node setting. Figure 12 shows the maximum throughput achieved by Storm and EDGEWISE with latency less than 100 ms. The scalability curve shows about 2.5x throughput improvement with 8 nodes, suggesting that a combination of networking costs and perhaps a non-optimal topology keep us from realizing the full 8x potential improvement. On this particular physical topology, EDGEWISE achieves an 18–71% improvement over Storm's maximum throughput, comparable to the 57% improvement obtained in the 1-node experiment.

8 Related Work

To the best of our knowledge, Edgent [12] is the only other SPE tailored for the Edge. Edgent is designed for data preprocessing at individual Edge devices rather than

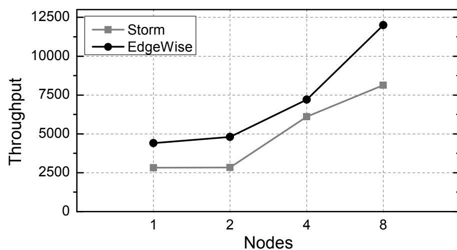

Figure 12: The maximum throughput achieved with the latency less than 100 ms, using the PRED topology with 1, 2, 4, and 8 distributed nodes (log-scale). EDGEWISE's intra-node optimizations extend to a distributed setting.

full-fledged distributed stream processing. We believe the Edge is powerful enough for more intelligent services, and thus EDGEWISE targets a more expressive SPE language with balanced throughput and latency.

Targeting distributed Cloud settings, a variety of academic [26, 39, 52, 62, 73] and industry/open-source [15, 49, 66] SPEs have been proposed. Some [8, 9, 11, 13, 14] support both stream and batch processing. Most use the OWPOA design and differentiate themselves on large-scale distributed processing and fault tolerance. None has our notion of an engine-level scheduler.

Researchers have studied distributed "job placement" schedulers for Storm [10, 23, 60, 77] and Spark Streaming [47, 51]. They determine where to distribute computing workloads across nodes in the Cloud. EDGEWISE does not focus on how to partition a physical topology for distributed computing.

Recent single-node SPE solutions leverage modern many-core and heterogeneous architectures. For example, GStream [78] describes an SPE tailored to GPUs, while Saber [48] is a hybrid SPE for mixed CPU/GPU computing that schedules operations on different hardware depending on where they perform better. StreamBox [56] explores the high-end server space, demonstrating high throughput by extracting pipeline parallelism on a state-of-the-art 56-core NUMA server. StreamBox uses a worker pool (like EDGEWISE) to maximize CPU utilization, but does not maintain a queue for each operation, making it hard to apply the Storm-like distributed stream processing model. EDGEWISE targets the opposite end of the spectrum: a compute cluster composed of Edge-class devices where intra-node scale-up is limited but inter-node scale-out is feasible. Thus, EDGEWISE adopts the general distributed streaming model (scale-out) and enables additional intra-node scale-up.

The Staged Event Driven Architecture (SEDA) [75] was proposed to overcome the limitations of thread-based web server architectures (e.g., Apache Httpd [1]): namely, per-client memory and context-switch overheads. EDGEWISE shares the same observation and proposes a new congestion-aware scheduler towards a more efficient EDA, considering how to stage (schedule) stream processing operations.

Mobile Edge Computing (MEC) [42, 59] uses mobile devices to form an Edge. Unlike EDGEWISE, they focus on mobile-specific issues: e.g., mobility, LTE connections, etc. Two works support stream processing on mobile devices. Mobile Storm [57] ported Apache Storm as is. MobiStreams [74] focuses on fault tolerance when a participating mobile disappears. On the other hand, Mobile-Cloud Computing (MCC) [30, 34, 37, 45, 63] aims to offload computation from mobile to the Cloud. There will be a good synergy between EDGEWISE and MEC/MCC. EDGEWISE could be viewed as a local cloud to which they can offload computations.

Lastly, queueing theory has been used to analyze stream processing. The chief difference between previous analyses and our own lies in the assumption about worker sharing. In traditional queueing theory, each operation is assumed to fairly share the workers. For example, Vakilinia et al. [71] uses queueing network models to determine the smallest number of workers under the latency constraint, under the assumption that workers are uniformly shared among operations. The same is true for the analyses of Mak et al. [54] for series-parallel DAG and Beard et al. [19] for heterogeneous hardware. On the other hand, our analysis identifies the benefit of a non-uniform scheduling weight, assigning more workers to heavy-loaded operations.

9 Conclusion

Existing stream processing engines were designed for the Cloud and behave poorly in the Edge context. This paper presents EDGEWISE, a novel Edge-friendly stream processing engine. EDGEWISE improves throughput and latency thanks to its use of a congestion-aware scheduler and a fixed-size worker pool. Some of the ideas behind EDGEWISE were proposed in the past but forgotten by modern stream processing engines; we enhance these ideas with a new scheduling algorithm supported by a new queuing-theoretic analysis. Sometimes the answers in system design lie not in the future but in the past.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Dilma Da Silva, for their thoughts and guidance. This work was supported in part by the National Science Foundation, under grant CNS-1814430.

References

[1] The apache http server. http://httpd.apache.org.

[2] EdgeX Foundry. https://www.edgexfoundry.org/.

[3] Ispout api documentation. https://storm.apache.org/releases/1.1.2/javadocs/org/apache/storm/spout/ISpout.html.

[4] Kafka - A distributed Streaming Platform. https://kafka.apache.org/.

[5] OpenFog. https://www.openfogconsortium.org/.

[6] Storm benchmark. https://github.com/intel-hadoop/storm-benchmark.

[7] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. The design of the borealis stream processing engine. In *Cidr*, volume 5, pages 277–289, 2005.

[8] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: fault-tolerant stream processing at internet scale. *Proceedings of the VLDB Endowment*, 6(11):1033–1044, 2013.

[9] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8:1792–1803, 2015.

[10] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive online scheduling in storm. In *Proceedings of the 7th ACM international conference on Distributed event-based systems*, pages 207–218. ACM, 2013.

[11] Apache. Apache beam. https://flink.apache.org/.

[12] Apache. Apache Edgent - A Community for Accelerating Analytics at the Edge. https://edgent.apache.org/.

[13] Apache. Apache flink. https://flink.apache.org/.

[14] Apache. Apache spark. a fast and general engine for large-scale data processing. http://spark.apache.org/.

[15] Apache. Apache storm. an open source distributed realtime computation system. http://storm.apache.org/.

[16] Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Itaru Nishizawa, Justin Rosenstein, and Jennifer Widom. Stream: The stanford stream data manager (demonstration description). In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 665–665, New York, NY, USA, 2003. ACM.

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[18] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, 13(4):333–353, 2004.

[19] Jonathan C Beard and Roger D Chamberlain. Analysis of a simple approach to modeling performance for streaming data applications. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 345–349. IEEE, 2013.

[20] Sean Boland. Ranking popular distributed computing packages for data science, 2018.

[21] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, 2012.

[22] Data Canvas. Sense your city: Data art challenge. http://datacanvas.org/sense-your-city/.

[23] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed qos-aware scheduling in storm. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, pages 344–347. ACM, 2015.

[24] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: a new class of data management applications. In *Proceedings of the*

28th international conference on Very Large Data Bases, pages 215–226. VLDB Endowment, 2002.

[25] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 838–849. VLDB Endowment, 2003.

[26] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[27] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[28] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stanley B Zdonik. Scalable distributed stream processing. In *CIDR*, volume 3, pages 257–268, 2003.

[29] Sanket Chintapalli, Derek Dagit, Bobby Evans, Reza Farivar, Thomas Graves, Mark Holderbaugh, Zhuo Liu, Kyle Nusbaum, Kishorkumar Patil, Boyang Jerry Peng, et al. Benchmarking streaming computation engines: Storm, flink and spark streaming. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pages 1789–1792. IEEE, 2016.

[30] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[31] Cisco. Cisco Kinetic Edge & Fog Processing Module (EFM). https://www.cisco.com/c/dam/en/us/solutions/collateral/internet-of-things/kinetic-datasheet-efm.pdf.

[32] Rebecca L Collins and Luca P Carloni. Flexible filters: load balancing through backpressure for stream programs. In *Proceedings of the seventh ACM international conference on Embedded software*, pages 205–214. ACM, 2009.

[33] Mckinsey & Company. The internet of things: Mapping the value beyond the hype, 2015.

[34] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: Making smartphones last longer with code offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[35] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. Approximate join processing over data streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 40–51, New York, NY, USA, 2003. ACM.

[36] Eclipse. Eclipse kura. open-source framework for iot. http://www.eclipse.org/kura/.

[37] Mark S. Gordon, D. Anoushe Jamshidi, Scott Mahlke, Z. Morley Mao, and Xu Chen. Comet: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 93–106, Berkeley, CA, USA, 2012. USENIX Association.

[38] Donald Gross. *Fundamentals of queueing theory*. John Wiley & Sons, 2008.

[39] Vincenzo Gulisano, Ricardo Jimenez-Peris, Marta Patino-Martinez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, 2012.

[40] HortonWorks. Apache storm topology tuning approach. https://community.hortonworks.com/articles/62852/feed-the-hungry-squirrel-series-storm-topology-tun.html.

[41] HortonWorks. Hortonworks best practices guide for apache storm. https://community.hortonworks.com/articles/550/unofficial-storm-and-kafka-best-practices-guide.html.

[42] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11(11):1–16, 2015.

[43] International Electrotechnical Commission (IEC). Iot 2020: Smart and secure iot platform. http://www.iec.ch/whitepaper/pdf/iecWP-loT2020-LR.pdf.

[44] C. Jennings, Z. Shelby, J. Arkko, and A. Keranen. Media types for sensor markup language (senml). 2016.

[45] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 615–629, New York, NY, USA, 2017. ACM.

[46] Leonard Kleinrock. *Queueing systems, volume 2: Computer applications*, volume 66. wiley New York, 1976.

[47] Dzmitry Kliazovich, Pascal Bouvry, and Samee Ullah Khan. Dens: Data center energy-efficient network-aware scheduling. *Cluster Computing*, 16(1):65–75, March 2013.

[48] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[49] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 239–250. ACM, 2015.

[50] Franck L Lewis et al. Wireless sensor networks. *Smart environments: technologies, protocols, and applications*, 11:46, 2004.

[51] Zhen Li, Bin Chen, Xiaocheng Liu, Dandan Ning, Qihang Wei, Yiping Wang, and Xiaogang Qiu. Bandwidth-guaranteed resource allocation and scheduling for parallel jobs in cloud data center. *Symmetry*, 10(5):134, 2018.

[52] Wei Lin, Haochuan Fan, Zhengping Qian, Junwei Xu, Sen Yang, Jingren Zhou, and Lidong Zhou. Streamscope: Continuous reliable distributed processing of big data streams. In *NSDI*, volume 16, pages 439–453, 2016.

[53] Ruirui Lu, Gang Wu, Bin Xie, and Jingtong Hu. Stream bench: Towards benchmarking modern distributed stream computing frameworks. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 69–78, Washington, DC, USA, 2014. IEEE Computer Society.

[54] Victor W Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, 1990.

[55] Dennis McCarthy and Umeshwar Dayal. The architecture of an active database management system. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, pages 215–224, New York, NY, USA, 1989. ACM.

[56] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. Streambox: Modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, pages 617–629, Berkeley, CA, USA, 2017. USENIX Association.

[57] Qian Ning, Chien-An Chen, Radu Stoleru, and Congcong Chen. Mobile storm: Distributed real-time stream processing for mobile clouds. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 139–145. IEEE, 2015.

[58] A. Papageorgiou, E. Poormohammady, and B. Cheng. Edge-computing-aware deployment of stream processing tasks based on topology-external information: Model, algorithms, and a storm-based prototype. In *2016 IEEE International Congress on Big Data (BigData Congress)*, pages 259–266, June 2016.

[59] Milan Patel, B Naughton, C Chan, N Sprecher, S Abeta, A Neal, et al. Mobile-edge computing

introductory technical white paper. *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.

[60] Boyang Peng, Mohammad Hosseini, Zhihao Hong, Reza Farivar, and Roy Campbell. R-storm: Resource-aware scheduling in storm. In *Proceedings of the 16th Annual Middleware Conference*, pages 149–161. ACM, 2015.

[61] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. In *ACM SIGPLAN Notices*, volume 46, pages 291–304. ACM, 2011.

[62] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 1–14, New York, NY, USA, 2013. ACM.

[63] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[64] H. P. Sajjad, K. Danniswara, A. Al-Shishtawy, and V. Vlassov. Spanedge: Towards unifying stream processing over central and near-the-edge data centers. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 168–178, Oct 2016.

[65] Jerome H Saltzer, David P Reed, and David D Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[66] Scott Schneider and Kun-Lung Wu. Low-synchronization, mostly lock-free, elastic scheduling for streaming runtimes. *ACM SIGPLAN Notices*, 52(6):648–661, 2017.

[67] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, Oct 2016.

[68] Anshu Shukla, Shilpa Chaturvedi, and Yogesh Simmhan. Riotbench: a real-time iot benchmark for distributed stream processing platforms. *arXiv preprint arXiv:1701.08530*, 2017.

[69] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[70] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[71] Shahin Vakilinia, Xinyao Zhang, and Dongyu Qiu. Analysis and optimization of big-data stream processing. In *2016 IEEE global communications conference (GLOBECOM)*, pages 1–6. IEEE, 2016.

[72] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta N Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. Farmbeats: An iot platform for data-driven agriculture. In *NSDI*, pages 515–529, 2017.

[73] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J. Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 374–389, New York, NY, USA, 2017. ACM.

[74] Huayong Wang and Li-Shiuan Peh. Mobistreams: A reliable distributed stream processing system for mobile devices. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 51–60. IEEE, 2014.

[75] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[76] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[77] Jielong Xu, Zhenhua Chen, Jian Tang, and Sen Su. T-storm: Traffic-aware online scheduling in storm. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on*, pages 535–544. IEEE, 2014.

[78] Yongpeng Zhang and Frank Mueller. Gstream: A general-purpose data streaming framework on gpu clusters. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 245–254. IEEE, 2011.