Geo-Distributed Stream Processing

Caleb Stanford

February 2020

Abstract

Distributed stream processing systems (DSPS) are specialized software platforms for processing large quantities of data in real time. While such systems (e.g. Apache Storm, Apache Flink) have achieved widespread use, they generally rely on sending data to a central computing cluster or data center to be processed, which means there are fundamental limits to (1) the end-to-end latency and (2) the network bandwidth used to communicate with the data center, particularly when the data to be processed is highly geo-distributed.

In this report, we will survey some recent methods to extend DSPS to the geo-distributed setting by executing stream processing jobs over a geo-distributed network of nodes. We will present representative papers from three different approaches: formal optimization techniques [1], extensions to the programming model [2], and using approximation to greatly reduce the size or frequency of data that needs to be sent between nodes [3]. We will discuss how these approaches differ, how they compare in terms of performance, and areas where future research is needed.

1 Introduction

In the last 15 years, distributed stream processing systems (DSPS) have emerged as a way to conveniently express and efficiently deploy computations over streaming data. Notable examples of such systems include early streaming databases like Aurora [5] and Borealis [4], and more recently Apache Storm [20], Apache Flink [17], Apache Samza [18, 28], Apache Spark Streaming [19, 45], Twitter (now Apache) Heron [40], Google MillWheel [7], and Naiad [27]. The primary feature of stream processing systems as contrasted with batch processing systems is that, besides high throughput and fault tolerance, they additionally offer continuous processing of data with low latency.

All mainstream distributed stream processing systems, however, share a limiting assumption: they require that all data is sent to a central distributed data center or computing cluster to be processed. This implies that, first, although they can achieve latency in the milliseconds, this includes only the time from when an event enters the data center to when a response is produced; and the end-to-end latency of data being sent over the network may be much higher. Second, in cases where data is extremely high volume, such as in processing geo-distributed video streams, the bandwidth used to send all the data to a central location can be prohibitive. This is made worse because available bandwidth between geo-distributed sites can vary over time, and periods of low availability cause spikes in latency ([3], Figure 1). These limitations motivate the need for a new generation of stream processing systems which embrace the geo-distributed nature of data: they should enable processing data without sending it to a central cluster.

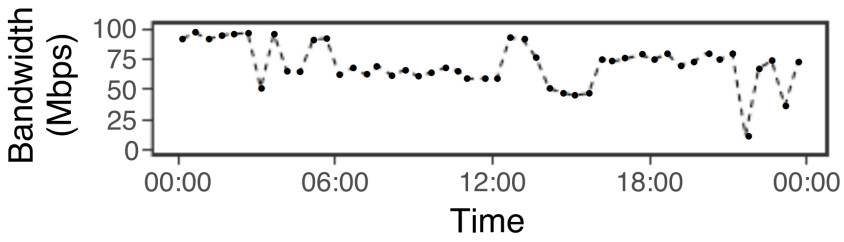

Figure 1: Available bandwidth over time between Amazon EC2 instances in Ireland and California. Chart is taken from [3].

Note that these limitations are not unique to distributed stream processing; the idea that systems should be aware of geo-distributed network conditions goes back to the long line of work in geo-distributed, WAN-aware, and more recently edge and fog computing systems. Another way of saying the problem is that in the DSPS domain, currently available solutions adopt the *cloud computing* paradigm. The vast amount of data that are being generated in the near future, particularly from video streams and IoT devices, are too high to be processed by today's networks. We seek to bring the ideas of *edge computing* [37] to the DSPS domain because it is a promising software platform for the emerging applications over this future data.

In this report, we investigate three recent proposed solutions to the geo-distributed stream processing problem. We have identified three categories of work in this domain. First, some papers have focused on *extending the system with optimization strategies* to account for geo-distributed data and nodes. This includes more sophisticated scheduling and resource placement. We discuss one promising such approach called Optimal Operator Placement [1], based on formalizing the placement problem as a mixed-integer linear programming (MILP) problem. Second, some papers have focused on providing new programming frameworks which allow better control over the placement and allocation of geo-distributed resources. We look at the system SpanEdge [2], which can be seen as a less automated, but much more lightweight version of the first approach. Finally, some papers have focused on the promising idea that, for many geo-distributed applications, a good way to deal with the vast amount of data and bandwidth constraints is to rely on *approximation*, by decreasing the quality or quantity of data sent, but in a way that retains a similar output. AWStream [2] is one recent such system, which also extends the programming model in an interesting way to express possible degradation functions, and compares them for which is the most accurate while also reducing bandwidth the most. In summary, we discuss the following three papers:

- Optimal operator placement for distributed stream processing applications. Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Distributed Event-Based Systems (DEBS) 2016. [1] (Section 3)
- SpanEdge: Towards unifying stream processing over central and near-the-edge data centers. Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir Vlassov. Symposium on Edge Computing (SEC) 2016. [2] (Section 4)
- AWStream: Adaptive wide-area streaming analytics. Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A. Lee. SIGCOMM 2018. [3] (Section 5)

Before discussing these papers, we provide some background in Section 2. Specifically, we explain what existing DSPS are in wide use and what they have in common, particularly with respect to the programming model. We use this to set up a common problem framework in which the geo-distributed stream processing problem can be studied, and we consider all three approaches as instances of this framework in Sections 3 through 5. Next, we survey other papers along the three categories in Section 6. Finally, we provide an overall comparison of the different directions, as well as promising directions for future work, in Section 7.

2 Background and Problem Framework

2.1 Background

Distributed Stream Processing Systems. There are a vast number of DSPS, including many research prototypes as well as actively developed software in widespread use. Some of the major systems are listed in Table 2. Different systems often differ in some details, particularly with regard to implementation choices (e.g. scheduling and optimization strategies, fault tolerance mechanisms) and sometimes with regard to the abstractions that are offered to the programmer. We focus in this report on the features that they share in common, rather than what distinguishes them, in order to provide a common framework to consider the geo-distributed streaming problem.

All DSPS in current use share the *dataflow programming model.* This means that the programmer writes, in some form or another, a dataflow graph. In some systems, e.g. Apache Storm, the dataflow graph is written out explicitly, whereas in others, such as Apache Flink, the graph is implicit. Additionally, systems may offer high-level libraries for creating or composing dataflow graphs; in particular, these include libraries for complex windowing operations and for SQL- and CQL-based streaming queries. The dataflow programming model exposes task and pipeline parallelism; to expose additional data parallelism, DSPS use *operator replication.* Similar to how a MapReduce [15] job is implicitly parallelized, all operators in a dataflow grpah (unless configured otherwise) may be split into several copies; this is part of the programming model as well, and affects the semantics.

The Dataflow Programming Model. We introduce the programming model through a simplified example based on a geo-distributed real-time video analytics use case. Imagine a large-scale system of video cameras, perhaps located in several cities throughout a country. Each video camera produces a stream of video data, at a certain frame rate and image resolution. Suppose that we want to identify pedestrians and report to a central location the summary of all pedestrian activity in the last 10 minutes, i.e., where pedestrians are most active. To do so, we want to classify each image from each camera using an out-of-the-box classifier; then, to prevent noise and to summarize the total activity, we want to aggregate the data from all classifiers in the last 10 minutes in a particular location (e.g., one intersection or group of intersections). In the end, we report for each location the total amount of pedestrian activity. (One could imagine taking further steps, such as adding a smoothing filter which removes reports of pedestrian activity that last only for a single frame, assuming that these must be erroneous.)

A dataflow pipeline for this is given in Figure 3. The input data consists of raw video data. Notice that the pipeline contains only one operator at each stage; that is, we treat

System	Year	Stable Release	Active?	Questions on StackOverflow (as of 2020-02-04)
Aurora	2003 [5]	2003 [29]	No	–
Borealis	2005 [4]	2008 [30]	No	–
 [32]	2011	2019 [20]	Yes	2458
 Apache Flink [17, 10]	2011	2019	Yes	3667
Google MillWheel	2013 [7]	–	No	–
 (Apache Spark Streaming)	2013 [45]	2019 [19]	Yes	4744
 [18, 28]	2013	2019	Yes	79
Naiad	2013 [27]	–	No	–
 [40, 24]	2015	2019	Yes	43

Figure 2: A selection of major distributed stream processing systems.

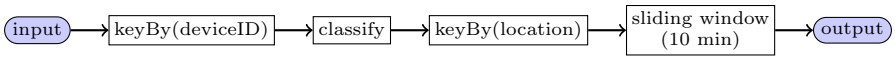

Figure 3: Example DSP dataflow graph for a program to classify video streams and report total pedestrian activity.

all input data at *all* cameras as a single input stream; and we write transformations over that stream. The first transformation, `keyBy(deviceID)`, says to divide the stream into substreams for different keys, where each key is a device ID. (Note that in some frameworks, such as Apache Storm, this would not necessarily be an explicit operator in the dataflow, but rather an annotation on the input data streams.) The second transformation, `classify`, says to process each input data item, which is a video frame, and return the classification (1 for a pedestrian, 0 for no pedestrian). The third transformation is similar to the first, but this time we group by location, instead of by device ID. The final transformation is the sliding window which adds up all the values over the last 10 minutes. This is then output and could be displayed to the user as a real-time report of activity across all locations.

In general, streaming dataflow graphs are acyclic, although some systems support ways to provide feedback and/or support iterative (cyclic) computations. The input and output nodes in a dataflow are special, because they interact with the external system, and can usually be of several kinds: e.g. taking input from a distributed file system, taking input from a live stream of data, writing output to a file, or in general interacting with some external service which provides input or consumes output. It is generally preferred that *internal nodes do not consume input or produce output*; however, that does not mean they are pure; they are often stateful, and may also produce logs, interact with a stored database, etc. Formally, we summarize a definition of a streaming dataflow graph that is (approximately) common to all DSPS programming models.

Definition 1 (Streaming Dataflow Graph)**.** An *acyclic streaming dataflow graph* (DSP dataflow graph) consists of a set of input nodes called *sources*, a set of intermediate nodes called *operators*, and a set of output nodes called *sinks*, connected by a set of directed edges which are *data streams*. The nodes and edges form a directed acyclic graph (DAG). Each source node may produce data items continuously, and each sink node consumes data items.

Operators are possibly stateful streaming functions that describe how to process an input item and when to produce output. They do not get to choose which of their input streams to read from; rather they have an input even handler which can be called on any event when it arrives. They may produce any number of outputs on any input item, in any combination of output streams.

The number of output items produced per input item is often called the operator's *selectivity*, which may be e.g. 1 (for a map), $0-1$ (for a filter), or more than 1 (for a copy). The fact that the selectivity is not constant is a major challenge that makes scheduling DSPS applications more difficult.

Auto-Parallelization (Operator Replication). DSPS rely on three types of parallelization to achieve scalability, especially to achieve high throughput. The first two are explicitly exposed in any acyclic streaming dataflow graph. *Pipeline parallelism*, which is visible in Figure 3, means that different operators in a sequential pipeline can be run by different workers, threads, or distributed nodes. *Task parallelism* is not shown in our example, but means that different operators in a parallel set of disjoint tasks (parallel nodes in

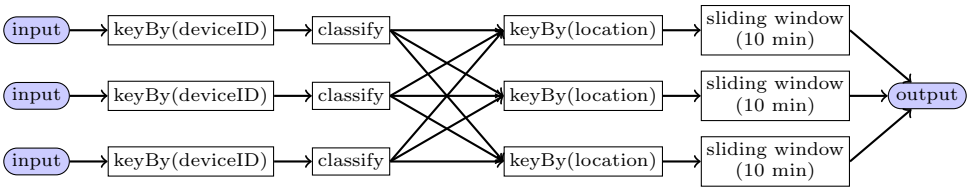

Figure 4: Example DSP dataflow graph after operator replication.

the dataflow) can be run by different workers, threads, or distributed nodes. However, the arguably most important form of parallelism for huge data sources is *data parallelism*, where different data items in the same input stream are processed by different workers, threads, or distributed nodes. DSPS use *auto-parallelization* to accomplish data parallelism, and unlike the other two kinds of parallelism, it modifies the dataflow graph and potentially the semantics of the program.

In our example of Figure 3, we want to exploit data parallelism on video streams from different cameras. In the `classify` stage of the pipeline, we are classifying images from different cameras separately, so it should be able to be run in parallel. The problem with data parallelism is that it would be cumbersome for the programmer to expose on their own; they would be forced to explicitly write dozens or hundreds of copies of the `classify` operator, and manually divide the source into dozens or hundreds of different sources, so that each operator got its own subset of the input data. To avoid this, DSPS automatically replicate operators in the dataflow graph into several parallel copies. Typically, the number of parallel copies can be configured by the programmer by setting the level of parallelism.

On our example, a possible auto-parallelized graph produced by the system is shown in Figure 4. Each operator is replicated, in this case into 3 copies. However, this does not fully describe what happens, because we have to understand the connections between operators. If there was an edge before, now there are 9 possible edges, and the system must decide which of them to use, and how to send data along the edges.

Typically there are several possible strategies, and the system chooses one based on context and/or explicit user configuration. One strategy is *round-robin*, where outputs from one stage of the pipeline are sent to the inputs of the next stage in round-robin order (so for every 3 outputs from one stage, 1 gets sent to each of the 3 operators in the following stage). A second strategy is *key-based partitioning* where the operator copy that an item is sent to depends on (a hash of) a specified key. Finally, in cases where there are the same number of parallel copies from one stage to the next, it is common to *preserve the same partitioning* from one stage to the next.

The partition operator `keyBy` in our example dataflow graph has no effect except to force a particular strategy for connections between stages: key-based partitioning based on the specified key. In Figure 4, first we assume that the input arrives partitioned by device ID. The `keyBy` by device ID then preserves such a partitioning. All future operators preserve the same partitioning, except when there is a second `keyBy` by location – this operator re-partitions the data based on location instead of device ID.

Formally, we make no assumption about the different DSPS programming models and how they handle connections between stages, except that they should obey any constraints that are explicitly programmed by the user. We capture the allowable parallelization in an annotated DSP graph in the following definition.

Definition 2 (Annotated Dataflow Graph and Parallelization)**.** An *annotated dataflow graph* consists of a dataflow graph annotated with, for each edge, whether the connection between parallel operator replicas should be (1) round-robin, (2) on the basis of a particular key, (3) partition-preserving, or (4) unspecified. Additionally, each vertex may be labeled with a level of parallelism that is allowed (1 for no parallelism).

A *parallelization* of an annotated dataflow graph consists of a larger graph, where: (i) each vertex is replicated to between 1 and i copies, where i is allowed level of parallelism; (ii) for each edge between v and v', if v has i copies and v' has i' copies, the connection must be consistent with the annotation. Specifically, (1) for round-robin items should be assigned to each operator in turn; (2) for key-based partitioning there should exist *some* partitioning of the keys such that that partition determines where a data item is sent next; (3) for partition-preserving, we must have $i = i'$ and the connections are one input to one output; (4) for unspecified, each output item from one stage may be sent to any of the operators for the next stage, as the system sees fit.

Distributed Runtime (Fault Tolerance and Scheduling). So far, we have explained how the programming model works in DSPS, in order to set the stage for the geo-distributed streaming problem. What we have described so far (having a dataflow graph which is parallelized by replicating operators) is enough to understand the geo-distributed streaming problem; the question will be how to execute these operators and avoid network communication overheads. However, since the programming model is only one small aspect of DSPS, we will explain the other implementation details and concerns that systems usually solve.

Once given a parallelized dataflow graph, the primary job of a DSPS runtime is to *schedule* workers in a distributed cluster so as to execute all the operators, continuously and in parallel. The goal of the scheduler is to maximally utilize the available distributed resources, and to prevent one task from becoming a performance bottleneck (called a *straggler*). In the case of stragglers, there are common techniques for the system to respond, e.g. *throttling* and *back-pressure*.

The performance of the system is generally measured in terms of *latency* and *throughput*. Latency is the time it takes for an output item to be produced after the input item which triggered the production arrives in the system. Throughput is total number of input items successfully processed per unit time. Throughput is often measured as *max throughput*, the maximum input rate the system can handle before it breaks. Given fixed resources, there is always some limit to how much input the system can handle; so max throughput is always finite. Beyond the max throughput, DSPS offer few guarantees about behavior, and will likely suffer linearly increasing latency, drop data, or crash altogether.

Finally, DSPS runtimes try to provide all of this with a guarantee of *fault-tolerance*. Whenever a worker is assigned to process some set of items from the input stream, the worker may fail. If this occurs, the system must have a way to rewind back to a safe state and re-process those input items, or re-assign them to a new worker. It is challenging to accomplish this in a way that minimizes overhead and also minimizes the time to recovery when experiencing a fault. See [42] for an example of a recent system that tries to attack the problem of balancing fast recovery time and low latency.

2.2 The Geo-Distributed Streaming Problem

As stated in the introduction, DSPS are built to handle the input data within the context of a data center or cluster. In particular, the latency that is optimized for is usually computed

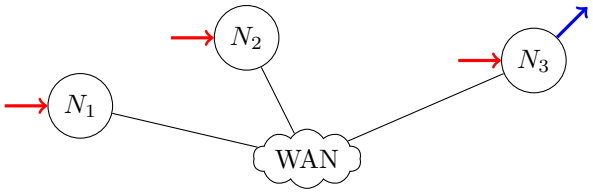

Figure 5: Example target topology: geo-distributed input streams (red), nodes (N_i), and output consumers (blue).

as the time between when an input item enters the system, and the time that the output is observed. This is an inadequate measure when the input and output are far away from the cluster; the actual latency could, in fact, be much higher. In addition, network conditions in the wide area are often such that high-bandwidth data, such as video streams, simply cannot be feasibly sent over the network in large quantities.

We consider in this report the problem of geo-distributed streaming. Given a dataflow graph as described in the previous section, assuming that it is annotated and will be executed in parallel, find a way to execute it on a geo-distributed set of nodes so as to minimize not just throughput and latency in the cluster, but total end-to-end latency as well as network bandwidth.

For instance, consider the example in Figure 3 and Figure 4. Suppose that the input video streams are in three different cities, corresponding to the three `input` source nodes in the parallel dataflow graph. And suppose there are three possible data centers (geo-distributed nodes) that can be used. This is captured by the following definition, and illustrated in Figure 5.

Definition 3 (Target Topology)**.** A *target topology* consists of (1) a finite set of geo-distributed nodes (single servers, clusters, or data centers); (2) a set of high-rate source input streams where each arrives into the system via one of the geo-distributed nodes; and (3) a set of output streams, or output consumers, which are also located at the geo-distributed nodes.

With traditional DSPS, one of the three datacenters (N_1, N_2, or N_3) will be selected to execute the query; and all three source video streams will be forwarded to that node. However, this results in high bandwidth when transferring the raw data stream over the network. The fix is simple: we should perform the `classification` step before sending all the data to a central location. That way, we only send the results of the classification, which are much lower bandwidth. In general, we (roughly) state the geo-distributed streaming problem in the following definition: the problem is to figure out how to compute the answer to the query while utilizing all the geo-distributed nodes and considering the constraints of latency and network bandwidth.

Definition 4 (Geo-Distributed Streaming Problem)**.** The system consists of a set of geo-distributed nodes, connected over the wide-area network (WAN). The problem is to execute a programmer-specified dataflow graph (Definition 1), parallelized through operator replication as desired (Definition 2). The sources in the dataflow graph are raw input data, e.g. video streams, arriving at various geo-distributed nodes. The output (sink) is consumed at specified geo-distributed nodes.

In the rest of this report, we consider three different approaches to the geo-distributed streaming problem. Some approaches will extend the programming model; all will modify the runtime in some way to make the execution geo-distributed.

3 Optimal Operator Placement

In this section, we consider the geo-distributed streaming problem introduced in Section 2 from the perspective of keeping the programming model as is; our goal is to instead extend the system implementation and optimization to the geo-distributed setting. In this paper [1], we do this in two steps. First, we *formally model* the geo-distributed setting, including the various constraints on latency and bandwidth. Second, we design a system implementation for solving these constraints to achieve *optimal* operator placement in the actual application. In the evaluation in Apache Storm, we will see how this approach compares with other baseline approaches, and how it scales.

The problem is formalized as a mixed-integer linear programming (MILP) problem. There are benefits and drawbacks of this approach. The benefit is that the formulation is not unique to network and bandwidth; it's very general, and can (and is) extended with various other metrics. For example, the authors consider resource constraints on the physical nodes, availability of nodes and links between nodes, and different constraints related to network bandwidth. All of these incorporate naturally into the framework. This was our primary interest in this paper, over all of the many other papers in this domain, which generally focus on a specific metric, and specific heuristics and solutions for those metrics. The drawback, of course, is that MILP is NP-complete, and the authors in fact show that their formulation of optimal operator placement is NP-hard. So the concern with this approach is primarily in how it will scale. However, we argue in the discussion that there are good reasons to believe this is a promising direction.

3.1 Operator Placement Formalization

We begin by formalizing the problem in our simple example. The dataflow graph we want to execute is the parallelized graph of Figure 4. Because the `keyBy` nodes are were just used to hint at how to apply parallelism, we can ignore them. So we are left with the following set of vertices and edges in the graph. For the vertices of the DSP graph, we have input streams s_1, s_2, s_3; `classify` nodes c_1, c_2, and c_3; `window` nodes w_1, w_2, w_3; and output o.

$$V_{dsp} = \{s_1, s_2, s_3, c_1, c_2, c_3, w_1, w_2, w_3, o\}.$$

For edges, we have

$$\begin{aligned} E_{dsp} = \{ \quad & (s_1, c_1), (s_2, c_2), (s_3, c_3), \\ & (c_1, w_1), (c_1, w_2), (c_1, w_3), (c_2, w_1), (c_2, w_2), (c_2, w_3), (c_3, w_1), (c_3, w_2), (c_3, w_3), \\ & (w_1, o), (w_2, o), (w_3, o) \quad \}. \end{aligned}$$

Together, these constitute the DSP graph. Next, we formalize the target topology. We have three nodes, N_1, N_2, and N_3, and we assume there are network links between all of them (including self-links since a node may send messages to itself):

$$\begin{aligned} V_{top} &= \{N_1, N_2, N_3\} \\ E_{top} &= V_{top} \times V_{top}. \end{aligned}$$

The problem will be to find an assignment of the vertices of the DSP graph to vertices of the topology (and this will imply an assignment of the edges of the DSP graph to the edges of the topology). Additionally, some nodes in the DSP graph may have constraints on which nodes in the physical graph they may be placed at. In our context, this is most useful in order to fix the input and outputs: recall that our target topology implies that i_1 is at N_1, i_2 is at N_2, i_3 is at N_3, and o is at N_3. So the assignment must satisfy these constraints.

For the rest of the presentation, let us compare two example assignments, one which should be better for latency and bandwidth, and the traditional centralized solution.

$$s_1, s_2, s_3, c_1, c_2, c_3, w_1, w_2, w_3, o \mapsto N_3 \quad \text{(centralized assignment)}$$

$$\begin{aligned} s_1, c_1 &\mapsto N_1 \\ s_2, c_2 &\mapsto N_2 \\ s_3, c_3, w_1, w_2, w_3, o &\mapsto N_3 \quad \text{(geo-distributed assignment).} \end{aligned}$$

For our next step of defining network bandwidth use and latency for the assignment, we need some additional parameters describing how much data is sent between nodes in our DSP graph, and how quickly data is processed. Later, these additional parameters will be obtained through profiling the system at runtime.

Latency. First, we consider latency. We need to know, for each pair of nodes, the average network delay when sending data from one node to the other. We also need to know, for each operator in the dataflow graph, how much time it takes to process each input item. Let $d_{i,j} \geq 0$ denote the average network delay between N_i and N_j, and let R_i denote the time to process an element at i. Let's assume that the `classify` operations take 50 time units, `window` operations take 5 time units, and that the network delay is 100 between N_1 and N_2 and N_2 and N_3, but 200 between N_1 and N_3:

$$\begin{aligned} d_{1,1} = d_{2,2} = d_{3,3} &= 0\,\text{ms} \\ d_{1,2} = d_{2,3} &= 100\,\text{ms} \\ d_{1,3} &= 200\,\text{ms} \\ R_{s_1} = R_{s_2} = R_{s_3} &= 0\,\text{ms} \\ R_{c_1} = R_{c_2} = R_{c_3} &= 50\,\text{ms} \\ R_{w_1} = R_{w_2} = R_{w_3} &= 5\,\text{ms} \\ R_o &= 0\,\text{ms}. \end{aligned}$$

We should note that $d_{i,j}$ represents the average delay, not the exact delay for all items, and similar for R_i. Even so, we are making several implicit assumptions here, which are heuristics and not completely true in practice. For instance, this model does not account for the fact that some stream processing schedulers trade latency for throughput by grouping items into small "batches" – they wait until an input buffer fills, or a timeout is reached, before processing the buffer. In this case, it is unhelpful to assume that R_i is a fixed processing time; some more complex model should be used, maybe adding a parameter for the buffer size. A larger buffer size should increase latency linearly.

Another assumption that is present here are that the processing time is independent of the node on which an operator is executed, and independent of what other operators

are being executed. To address these, first we will assume that each node comes with a "speedup" S_i over the base processing time. Second, we will assume that each physical node, as well as each DSP operator, has a certain resource capacity; as long as the total capacity of the node is not exceeded by the DSP nodes placed on it, the system will perform up-to-speed. This is likely a simplistic model, but correct to a first approximation as long as the capacity of physical nodes is conservatively small. For this presentation, we will assume that $S_i = 1$, so we will not include it in the formalized optimization problem. We will also ignore capacities, assuming that N_i are data clusters with sufficient capacity to execute any number of workers.

The parameters R_i and $d_{i,j}$ allow us to determine the total latency (response time) along *any path in the DSP graph*: this is the sum of R_i for all vertices i in that path, and the sum of $d_{i',j'}$ for all physical links corresponding to adjacent vertices i, j in the path.

Let us see how our two placement solutions (centralized and geo-distributed) compare for latency. In the centralized solution, any path in the DSP from input to output contains a cost of 50 for the classification and 5 for the windowing; most links are free (0 delay), except the links from input s_1, s_2, or s_3 to the classification node c_1, c_2, or c_3. Of these links, the worst is the link from N_1 to N_3 since it costs 200. So the maximum possible delay will be along this path: s_1, c_1, w_1, o. The latency is the total time along this path. For the geo-distributed solution, the analysis is similar: any path pays for the c_i and w_j delays, but the most expensive path is the one that goes from N_1 to N_3. The point where the delay occurs is different, but the latency is the same.

$$\begin{aligned}
\text{Latency}_c &= R_{s_1} + d_{1,3} + R_{c_1} + d_{3,3} + R_{w_1} + d_{3,3} + R_o \\
&= 0 + 200 + 50 + 0 + 5 + 0 + 0 = 255\,\text{ms}. \quad \textit{(centralized)} \\
\text{Latency}_g &= R_{s_1} + d_{1,1} + R_{c_1} + d_{1,3} + R_{w_1} + d_{3,3} + R_o \\
&= 0 + 0 + 50 + 200 + 5 + 0 + 0 = 255\,\text{ms}. \quad \textit{(geo-distributed)}
\end{aligned}$$

Notice that the two solutions are the same because in this case, the decisions are required at N_3, so there is no way to avoid sending the data to N_3 at some point. What will be saved, then, in the geo-distributed solution is mainly the network bandwidth. Additionally, our model does not account for the fact that, under a protocol like TCP, if bandwidth is constrained for a period time, latencies will start to grow and suffer during that time.

Bandwidth. For bandwidth, we can consider two types of constraints: hard constraints, where the bandwidth of a single link should never have more than a certain amount of traffic; and soft constraints, where we want to minimize the total amount of traffic (or a variant of traffic). In this presentation, we focus on soft constraints. In addition to R_i and $d_{i,j}$, we require one additional parameter: the data traffic rate $\lambda_{i,j}$ between nodes i and j in the DSP graph. This is measured in, say, Mb/s of data that are sent along a given edge in the DSP graph. Since raw video data is expensive while the simple label after classification is not, let us say that λ are as follows:

$$\begin{aligned}
\lambda_{s_i,c_i} &= 1000\,\text{Mb/s for all } i \\
\lambda_{c_i,w_j} &= 1\,\text{Mb/s for all } i, j \\
\lambda_{w_i,o} &= 1\,\text{Mb/s for all } i.
\end{aligned}$$

There are two soft constraints of interest: the total traffic between nodes, i.e. the amount of data communicated per second, and the total network usage between nodes, i.e.

the average amount of data that is in transit at a given time. Notice that larger delays between nodes do not affect total traffic, but they do affect total network usage. Formally, traffic is the sum over all DSP edges of $\lambda_{i,j} \cdot [i' \neq j']$, where $N_{i'}$ and $N_{j'}$ are the physical nodes that DSP vertices i and j are assigned to, respectively. That is, we add up all the λ traffic, but only include traffic between distinct nodes, not traffic within the same node. And network usage is the sum over all DSP edges of $\lambda_{i,j} d_{i',j'}$, where i' and j' are as before. Let us see how these metrics compare in our example assignments:

$$\begin{aligned}
\text{Traffic}_c &= \lambda_{s_1,c_1} + \lambda_{s_2,c_2} \\
&= 2000\,\text{Mb/s} \quad \textit{(centralized)} \\
\text{Traffic}_g &= \lambda_{c_1,w_1} + \lambda_{c_1,w_2} + \lambda_{c_1,w_3} + \lambda_{c_2,w_1} + \lambda_{c_2,w_2} + \lambda_{c_2,w_3} \\
&= 6\,\text{Mb/s} \quad \textit{(geo-distributed)} \\
\text{Network Usage}_c &= \lambda_{s_1,c_1} \cdot d_{1,3} + \lambda_{s_2,c_2} \cdot d_{2,3} \\
&= (1000\,\text{Mb/s}) \cdot (200\,\text{ms}) + (1000\,\text{Mb/s}) \cdot (100\,\text{ms}) \\
&= 300\,\text{Mb} \quad \textit{(centralized)} \\
\text{Network Usage}_g &= \lambda_{c_1,w_1} d_{1,3} + \lambda_{c_1,w_2} d_{1,3} + \lambda_{c_1,w_3} d_{1,3} + \lambda_{c_2,w_1} d_{2,3} + \lambda_{c_2,w_2} d_{2,3} + \lambda_{c_2,w_3} d_{2,3} \\
&= 3 \cdot (1\,\text{Mb/s}) \cdot (200\,\text{ms}) + 3 \cdot (1\,\text{Mb/s}) \cdot (100\,\text{ms}) \\
&= 0.9\,\text{Mb}. \quad \textit{(geo-distributed)}
\end{aligned}$$

As expected, the network traffic and usage are greatly reduced under the geo-distributed assignment.

Resulting Mixed Integer Linear Program. It is assumed that all the parameters ($d_{i,j}$, R_i, and $\lambda_{i,j}$) are known constants. Then, the formalization is put together into a mixed integer linear program (MILP) that can be solved to find the true optimal placement. The following are some technical details to make the translation work. First we need to describe a real-valued quantity to optimize. We can optimize either latency or traffic or network usage, or any weighted linear combination of the three. Also, we want to normalize so that they are between 0 and 1, so we need to first solve (or know) the maximum of each individually. For simplicity of presentation, we will just assume that we minimize the sum of latency and bandwidth, where bandwidth is traffic (not network usage), and we do not normalize. Second, we need to include variables for the placement itself. This is encoded using Boolean variables: for each vertex i of the DSP and physical node i', $x_{i,i'}$ is 1 if i is placed at i', and 0 otherwise. We need to constrain the sum over i of $x_{i,i'}$ to be 1, so that i is placed only at one node. We also make corresponding variables $y_{(i,j),(i',j')}$ for when both i is assigned to i', and j is assigned to j'. These have to be constrainted so that $y_{(i,j),(i',j')} = x_{i,i} \cdot x_{j,j'}$ (without using multiplication, since that isn't linear). Third, recall that the latency is the *maximum over all paths* of the total delay along this path; we need to convert this to linear constraints by introducing a variable L, representing the latency, and making it greater than all paths. (We could also say it's exactly equal to at least one path, but this is unnecessary.)

Putting all of this together, we end up with the following linear program.

$$\textbf{Boolean variables: } x_{i,i'} \text{ and } y_{(i,j),(i',j')} \text{ for } i,j \in V_{dsp}, i',j' \in V_{top}$$

$$\textbf{Real variables: } L$$

$$\textbf{Minimize } L + N \textbf{ subject to:}$$

$$L \geq \sum_{k=1}^{n} R_{i_k} + \sum_{k=1}^{n-1} \sum_{i',j' \in V_{top}} y_{(i_k,i'),(i_{k+1},j')} d_{i',j'}$$

$$\text{for all DSP paths } i_1, i_2, \ldots, i_n$$

$$N = \sum_{i,j,i',j'} \lambda_{i,j} y_{(i,j),(i',j')} [i' \neq j']$$

$$\sum_{i'} x_{i,i'} = 1 \text{ for all } i$$

$$\sum_{i} x_{i,i'} = 1 \text{ for all } i'$$

$$x_{i,i'} = \sum_{j' \in V_{top}} y_{(i,j),(i',j')} \text{ for all } i,j$$

$$x_{j,j'} = \sum_{i' \in V_{top}} y_{(i,j),(i',j')} \text{ for all } i,j.$$

Other Cost Metrics. In the paper [1], there is another nontrivial cost metric of *availability*, which represents the probability that the service is online and not down. Each link and each node has an availability; this is then part of the optimization problem along with the bandwidth and latency. In general, the framework is quite extensible; it seems that many metrics can be included and formulated as part of the MILP optimization problem.

3.2 Implementation

A scheduler based on the MILP formalization above was implemented on top of Apache Storm. The implementation requires solving the following problems: (1) converting the DSP and physical nodes in Storm to a formal DSP graph and topology graph; (2) profiling the network and operators to calculate the parameters (i.e. $d_{i,j}$, R_i, and $\lambda_{i,j}$); and (3) solving the resulting MILP problem. Of these, (1) is specific to Storm, so we do not discuss it here. The solutions to (2) and (3) are out-of-the-box: profiling tools that have previously been developed by the authors to keep track of metrics in Storm; and a state-of-the-art MILP solver.

In principal, the choice could be made to solve OOP either statically before the application is deployed, or to update it dynamically. The implementation makes an intermediate choice between the two; OOP is solved once at the beginning, once after profiling for some period of time, and then once whenever the application execution is compromised through a worker failure.

The most interesting questions for evaluation are the following: first, how successful is the OOP formalization compared to alternate existing approaches for minimizing these metrics? Is it truly optimal or does modeling error cause it to fail in practice? Second, when does the MILP approach fail – does the problem blow up with larger graphs to become impossible?

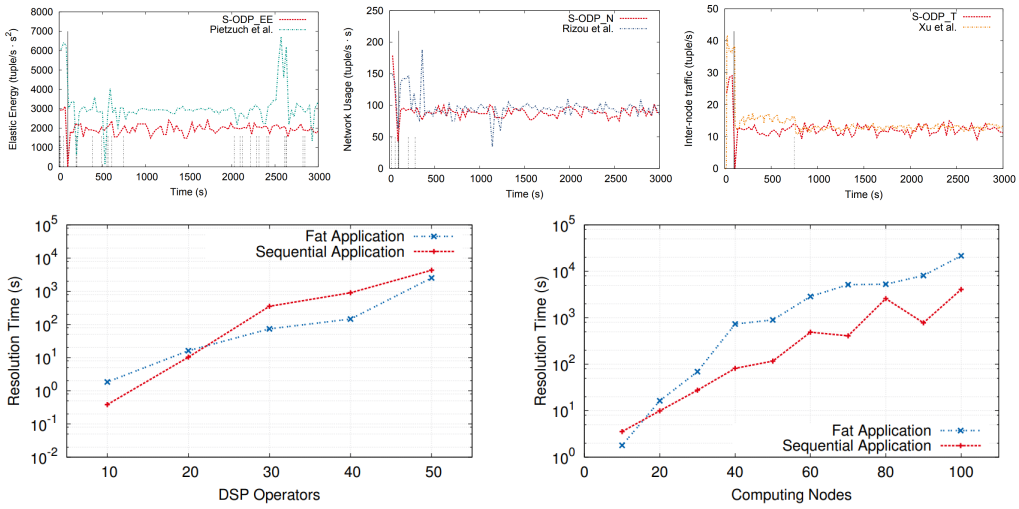

Figure 6: Key takeaways from the evaluation of Optimal Operator Placement in Storm. Charts are taken from [1].

We summarize some key takeaways related to these questions in Figure 6. First, the authors compare the OOP implementation with several solutions in the literature specific to different metrics: here, elasticity (a variant combination of network bandwidth and latency), network usage, and network traffic. The remarkable thing here is that OOP matches the optimal performance in all these cases; all that is required to tune OOP is to change the weights on what is optimized for. It would seem that, for these particular programs, there does not seem to be a modeling cost compared to the other solutions. Second, the authors consider scalability by creating a large DSP graph, either a long (sequential pipeline) application, or a fat (task-parallel) application. For 50 operators in the DSP, the system requires about 1000 seconds to find the optimal placement solution, and the scaling appears exponential. In the paper, the authors consider various ways that the search could be improved, and investigate the ability of the MILP to find a solution more quickly with different heuristics to simplify the search.

3.3 Discussion

Optimal operator placement seems to be a remarkably expressive framework. Not only latency and various metrics related to network bandwidth, the formalization can take into account availability and could be extended with many other cost metrics. The promise of this approach is that the system finds an optimal placement of DSP operators to nodes *with one uniform solution* to handle any desired cost metric: in the evaluation, it is shown that we can match good performance for various other solutions in the literature, which were proposed for specific cost metrics.

The biggest question in terms of limitations is the scalability, as solving the MILP problem could take exponential time in the worst case – and the evaluation seems to manifest this exponential blowup. It is likely that heuristic approaches are more efficient, especially for large graphs. However, there is hope that this approach is feasible. Firstly, if there is any domain where solving an expensive optimization problem could be worth it, this domain is

quite reasonable because the costs of input data are so huge. For instance, if solving the decision problem takes 5 or 10 hours, this is miniscule compared to the time to process days of live video streams, or compared to the total bandwidth to send all this video data over the network. It is likely that the incentives are such that solving the optimization problem is really worth it, and it does not need to be done frequently. Secondly, it is nice that the MILP solver (the costly part) is just a black-box a component that can be plugged in as desired. This means that, from a research perspective, work can be focused on making MILP solvers more effective and scalable. Finally, DSP graphs are user programs, so their size (or complexity from the perspective of the solver) may be bounded; they are not likely to be arbitrarily messy and difficult to reason about.

A second concern with this approach is the many ways that the formalization has been simplified. For instance, it doesn't seem correct to model the execution time of operators as a single real-valued delay. As we commented earlier, many DSP systems buffer items and delay processing them until either the buffer fills up, or a small timeout is reached. This should be accounted for in the model; this is a way of trading latency for throughput. It is likely that a perfect model is impossible. Could it be that in some cases where the model is imperfect, the MILP solving fails and the system behaves erratically? Also, in order to make the model more and more accurate, the problem could become arbitrarily complex. This becomes worse the more features are added to the implementation of Storm (or whatever other DSPS this is implemented in), as all additional features would ideally be incorporated in the model, at least if they have performance impact.

An interesting direction for future work would be to include throughput in the model. While throughput is one of the most important performance metrics for DSPS, it was notably missing from the formalization.

4 Programming Extensions: SpanEdge

In this section, we consider the geo-distributed streaming problem introduced in Section 2 with the following view: rather than try to solve the (difficult) optimization problem in the implementation, we should instead enable the programmer to achieve the optimal distribution manually. As we saw in Section 3, while the solution there achieved good performance for different metrics, the solver did not scale well with the number of nodes in the DSP graph to be executed. Our hope in this section is that, with a small amount of assistance from the user, we can make the optimization problem a lot easier.

To this end, we seek to define implementation hints (a form of programming abstraction) that are conceptually simple. That is, we hope that the programmer does not have to understand all the details of the implementation to achieve efficient code. (Otherwise, they could just stop using a DSPS altogether, and write low-level code.) But at the same time as being simple, we also want the hints to be maximally *useful* optimization, i.e. we want to give as much information as possible to the system about a good solution to the geo-distributed streaming problem for the user's specific application scenario.

The abstraction that is chosen by SpanEdge [2] has to do with *local* and *global* computations. The idea is that some intermediate computations (and outputs) can be done locally, without sending data to a central location, whereas others require globally aggregating data, and there is no way to avoid sending data. The problem will then be to design algorithms to deploy the DSP graph with local and global annotations. In the evaluation, SpanEdge is compared with a traditional central deployment in Apache Storm, and it is shown to greatly

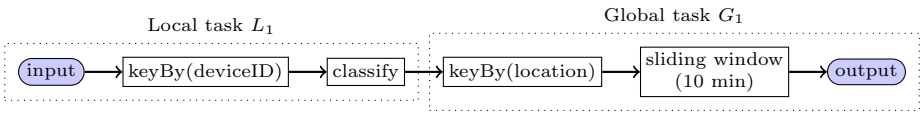

Figure 7: SpanEdge program for our running example: nodes in the dataflow graph are grouped into local and global tasks.

reduce network traffic and the latency required for local events, while not slowing down the processing of events that require global communication.

4.1 Programming model

To modify the programming model for DSPS to allow for geo-distributed stream processing, we will modify the dataflow graph of Definition 1. Note that the annotations have to be on the dataflow graph itself rather than the parallelized version, as in the previous section. So we can't simply ask the programmer to specify whether to execute each *parallel* operator at its source or centrally, as we don't know how many operators there will be after replication.

SpanEdge solves this with the notion of *local tasks* and *global tasks*. A task is a group of operators (or sources or sinks) in the DSP graph. Consider our example dataflow in Figure 3. Here we have 6 vertices to consider. The programmer considers that video streams are large and should be processed locally; therefore, the source node `input` should be local. Also, we want to do the classification locally, before sending anything to a data center, so `classify` can be local. Once we get to the `keyBy(location)`, it is safe to do this globally, and once we get the output we know this needs to be global. In summary, we want the first three nodes in the pipeline to be local, and the last three to be global. Notice that sources and sinks are part of these groupings as well, not just operators.

This division can be accomplished with a single local task L_1 and a single global task G_1. These are depicted in boxes, and shown in Figure 7 for our example. Concretely, in Storm, these annotations are very lightweight: the tasks can be assigned by putting an annotation on each individual node in the graph, specifically `.addConfiguration("local-task", "L1")` for the local nodes, and `.addConfiguration("global-task", "G1")` for the global nodes. In general, the DSP graph may have multiple sources and different computations on the different sources. This is where it would make sense to have multiple local tasks, one for each source and computations on that source. The tasks should not intersect, and there should be only one global task.

4.2 Implementation

The implementation of SpanEdge is done in Apache Storm, and compared with a centralized Storm deployment. The stated goals of the evaluation are to compare bandwidth usage and latency for these two solutions. For latency, note that there can be local and global results: for results that are produced and consumed locally, we expect that geo-distributed deployment leads to huge improvements in latency, whereas for results that require aggregating all global data (such as the windowing results in our running example of video stream pedestrian classification), we expect that there is negligible improvement. Thus, the evaluation of latency is divided into the latency for local results and for global results.

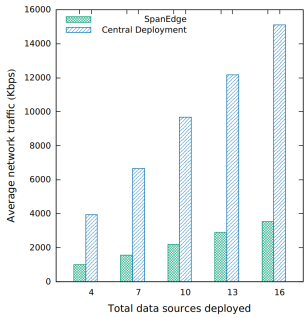

Fig. 8: The overall bandwidth consumption.

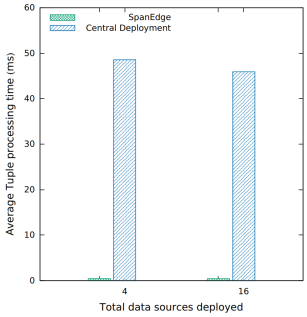

Fig. 9: The average tuple processing time (the local result).

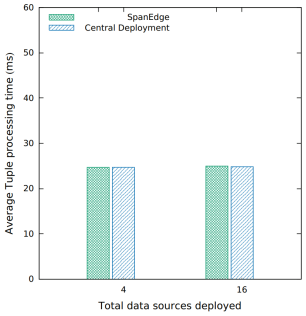

Fig. 10: The average tuple processing time (the global result).

Figure 8: Key takeaways from the evaluation of SpanEdge. Charts are taken from [2].

The results for these three metrics (network use, local latency, and global latency) are shown in Figure 8. For bandwidth savings, we see that although usage seems roughly linear for both SpanEdge and the baseline, SpanEdge uses perhaps 1/4 of the bandwidth. For local latency, we see even more impressive results, of less than a millisecond compared to 50 milliseconds in the central deployment. For global latency, as expected, there is no difference.

4.3 Discussion

The local/global grouping of nodes in the dataflow graph is a nice abstraction, that seems to be a good "sweet spot" between requiring minimal extra information from the programmer, while getting maximal benefit in the implementation. It seems fairly principled, in that there are well-defined semantics for the local and global tasks and this interacts well with the DSP model (there might be some constraints on when a task is allowed to be local and global that need to be enforced, but it is semantically compatible with the core notions of operator replication and parallelism).

The evaluation leaves room for improvement. Although SpanEdge shows excellent performance relative to a centralized execution in Storm, it is not compared to another geo-distributed solution; although the other two solutions in this report were not around at the time of publication, it would be interesting to see for instance how it compares with classic operator placement solutions like [33]. We expect that manual placement by labeling tasks as "local", if it is done correctly, would be more informative and thus lead to better performance; would this be true in reality? Another interesting thing to see would be how much it depends on the programmer to get it right. What is the performance cost if the programmer labels local and global tasks incorrectly?

Since the local and global grouping of tasks is such a simple concept, an interesting direction for future work would be to see if these groupings can be learned automatically. Compared to the complex (high-dimensional) and difficult (NP-complete) optimization problem from Section 3, choosing local and global tasks might be relatively easy. Can the optimal task assignment be found automatically by either a heuristic iterative search or a global exhaustive search? Finally, it would be interesting to see if there are other simple programming extensions which strike a similar balance and achieve good performance for other metrics, like throughput – e.g. labeling "expensive" or "bottleneck" operators in the dataflow graph, rather than having to manually increase the level of parallelism to some large integer.

5 Approximation via Degradation: AWStream

Unlike the previous approaches to the geo-distributed streaming problem introduced in Section 2, AWStream [3] relies on a different idea: approximation. Approximation is a common technique in programming streaming algorithms in general; for instance, when computing statistical summaries such as the median of a data stream. The idea of AWStream (and Jetstream [34] before it) is that high-bandwidth data streams between nodes can be *degraded* through some compression mechanism to reduce bandwidth use, while obtaining an approximately correct answer (maintaining high accuracy). This degradation can be applied selectively where needed in response to bandwidth constraints. Thus, the tradeoff explored here is between bandwidth use and accuracy.

AWStream is both an extension to the programming model and an extension to the system implementation and runtime. At the programming model level, the user specifies *possible* degradation operations, which are fully customizable. In the implementation, offline and offline profiling are used to determine how much the degradation operations improve bandwidth and how much they reduce accuracy; the system tries to find a balance between the two. Then, the degradation operations are applied whenever bandwidth constraints are detected as needed, that is they are applied dynamically. In other words, the system is *adaptive* to arbitrary changes in available WAN bandwidth. In contrast to the previous approaches which were deployed on top of an existing DSPS (Apache Storm), AWStream is implemented as a stand-alone system in Rust [31]. It is compared for streaming analytics applications (mostly on video data) against JetStream as well as against basic streaming over TCP and UDP.

5.1 Programming Framework

AWStream targets cases where network bandwidth cannot be avoided altogether through operator placement. Consider our example of pedestrian detection in video streams (Figure 3). Suppose, however, that we cannot avoid sending everything to a central location. There might be several reasons for this: nodes N_1 and N_2 might be resource-constrained; or the core classification logic might require input data from multiple video streams at once; or the video stream source is a "client" that should not know the classification logic, and the central data center is a "provider" that provides the classification when sent video data by the client. In this case, the previous solutions based on placing the `classify` operator near the data source do not work, but it should still be possible to reduce bandwidth through degradation.

To solve this, the programmer specifies possible *degradation functions* – ways to process the source data to decrease its size or quantity – using an abstraction called a *maybe knob*. A maybe knob states a degradation that might be performed, which is parameterized by a variable called a knob. The knob is given a finite number of possibilities. There are two basic example degradation operations on video streams: *decreasing frame rate* and *decreasing resolution*. Both of these would be maybe knobs: the knob in case of frame rate indicates how many frames to drop (e.g. drop 1 in 2 frames, 4 in 5 frames, or 9 in 10 frames); while the knob in case of resolution indicates how much to scale the image down by (e.g. cut width and height in 2, or in 3, or in 4). The pedestrian detection dataflow graph with these two maybe knobs added is shown in Figure 9.

The reason for specifying multiple possible degradation functions is that which one works well is highly application-specific: the authors show that for pedestrian detection, it is

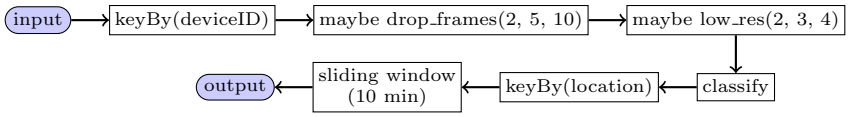

Figure 9: Example DSPS dataflow with AWStream's *maybe knobs* added for degradation.

important to keep high resolution so frame rate should be dropped instead, whereas for augmented reality, frame rate is more important and resolution should be degraded.

5.2 Implementation

The primary difficulty in the implementation of AWStream is how to monitor the network bandwidth use and quality under various degradation plans. In particular, the user-provided operations only specify what might be done, but do not come with predictable accuracy and bandwidth. AWStream uses a combination of both offline and online profiling. The offline profiling is expensive and blows up exponentially in the number of knobs for degradation, while the online profiling aims to be more lightweight and turns on only when there is spare bandwidth. However, expensive profiling can be required online as well: if a a huge difference occurs between the reality and the prediction of the profile, this triggers a full profiling.

The evaluation compares AWStream with other possible solutions for streaming video over a network: streaming over TCP (no response to limited bandwidth), streaming over UDP (limited bandwidth results in dropped packets), a baseline existing adaptive video streaming method (HLS), and JetStream. The bandwidth starts normal, and then is throttled during an intermediate period of shaping to see how the different solutions respond. Note that, for instance, TCP will just suffer increasing delays until it catches up with all missed packets after the shaping period; whereas UDP will suffer no drop in latency but will have no guarantee about accuracy. The results are in Figure 10. The application use case shown here is for an augmented reality video streaming application; there is also a pedestrian detection video streaming application, not shown. In summary, AWStream is the best solution to balance having near-perfect accuracy while optimally degrading the streams to make sure latency stays high during bandwidth shaping. Also note that, although HLS shows good accuracy in this example, it does not show good accuracy in the pedestrian detection example, because of the kind of data degradation that it performs.

The paper evaluates a number of other things, as well, particularly the cost of profiling techniques. It shows how sampling is used to vastly speed up online profiling against a baseline which would be $52x$ overhead; however the AWStream solutions are still $6x$ or $17x$ overhead. Also, AWStream is shown to maintain good accuracy and latency under various network delays.

5.3 Discussion

AWStream is conceptually and empirically appealing. The programming of "knobs" is straightforward, and the evaluation shows impressive ability to balance latency and accuracy during bandwidth shaping. It would be great to see more work in the DSPS domain in which approximation and data stream degradation play bigger roles; this seems like a promising way to scale these applications more generally, not just with regard to bandwidth usage.

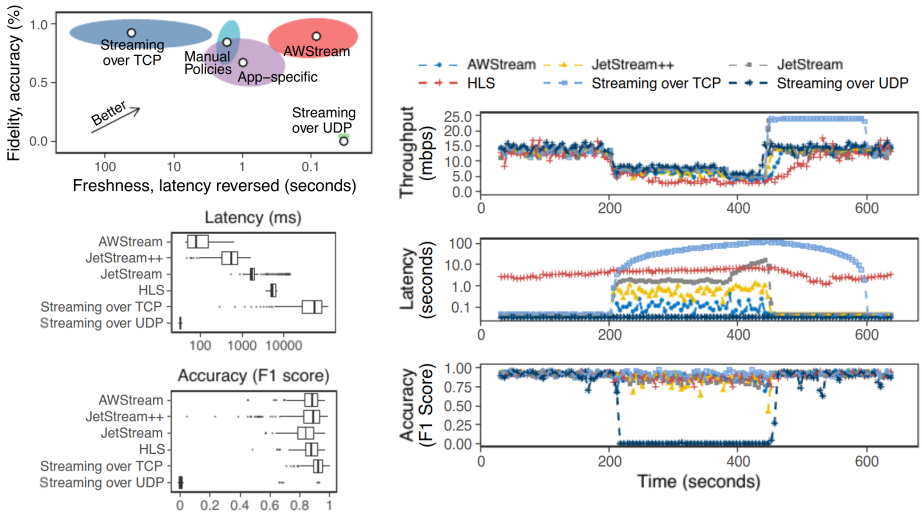

Figure 10: Key takeaways from the evaluation of AWStream. Charts are taken from [3].

The primary drawback of AWStream as it was presented is probably the overhead of the offline and online profiling. The space of possible degradations to explore is exponentially large, because each combination of a setting for every knob has to be considered. And although the authors have taken great effort to improve the overhead of online monitoring over a continuously monitoring baseline, there remains at least a $6x$ overhead, which seems highly significant. On the other hand, profiling can be avoided in the long term after accurate profiles are known: so similar to the high cost of solving the optimization problem that we discussed in Section 3, these high profiling costs may not actually be significant compared to the amount of data processed by the system over a period of days, weeks, or months. This depends on the extent to which online profiling can be disabled, which isn't totally clear.

AWStream does not incorporate operator placement, but rather the dataflow graph is split into computations on the source data, and computations at a server (somewhat like the local/global division in SpanEdge in Section 4). It would be interesting to combine operator placement with approximation and degradation. It would also be interesting to see how AWStream compares with operator placement solutions (not just how it compares with JetStream). Finally, there is room for exploration in the design of more high-level and expressive programming models for expressing degradation functions.

6 Related Work

These three papers are not the only ones to tackle the geo-distributed streaming problem. We have selected them because we found them to be representative, and because they were some of the more illuminating approaches in terms of clear research methodology and possible future work. Let us briefly discuss other work that we are aware of in these three categories (this list is not exhaustive).

Network-Aware Operator Placement. Other than [1], there are several lines of work in job scheduling, operator placement, and optimization for DSPS that try to be network-aware in some fashion. Early and influential works include [6] and [33]. The first [6] is probably the first to formalize the operator placement problem for DSPS, and to explore (1) how network-awareness can lead to more efficient query evaluation, and (2) how there is a trade-off between latency and bandwidth use in this space. The second [33] presents a simple but effective heuristic algorithm which treats the geo-distributed physical nodes as a system of points in a combined latency-bandwidth space, and uses spring relaxation find a good configuration. The work [22], similarly to [1], formulates the problem in MILP. There are a number of other related papers on job scheduling [8, 44, 16, 43, 21], operator placement [9, 41, 36, 25], and resource elasticity [12, 23, 11, 14].

Programming Frameworks. A system very related to SpanEdge [2] is Geelytics [13]. It modifies the dataflow programming model with *scoped tasks*, which have a geographic granularity such as by site, by city, by district, or by section, and are similar to SpanEdge's local and global tasks. Other than these, the paper [35] proposes a programming framework for stream processing in a geo-distributed (at the edge) fashion. The programming framework, however, is not based on dataflow, and is more focused on the communication mechanisms between nodes. There is also a large body of work on programming for wireless sensor networks; see the survey [26]. In general, the concerns in that domain have been more low-level, related to connections between sensors, mobility of sensors, communication from one sensor to another via short hops, and so on.

Systems Incorporating Approximation and Degradation. The first stream processing system to incorporate the degradation of data streams as found in AWStream [3] was JetStream [34]. JetStream seeks to limit bandwidth use not just through degradation, but also "aggregation", which refers to a data model where data is saved and aggregated by geo-distributed nodes, and only sent when explicitly requested. Besides this, an early idea in DSPS that predates degradation is *load shedding* [38, 39]. This refers to selectively dropping data in response to load that is beyond capacity, in order to maintain availability and good latency while hopefully not losing too much accuracy.

7 Conclusions

We have discussed three different approaches to executing a DSP dataflow graph over geo-distributed nodes and source streams: one which formalizes the problem as a MILP, one which extends the programming model to allow tasks to be executed locally, and one which relies on approximation by automatically degrading data streams when network bandwidth is insufficient. A high-level comparison of the approaches is in Figure 11. Optimal Operator Placement is the only solution that is purely optimization, i.e. requires no changes to the programming model. In terms of overhead, Optimal Operator Placement and AWStream require large static overheads to compute the best solution, and both require some profiling overhead to obtain information about the running system. SpanEdge is the most lightweight solution, requiring very little overhead. On the other hand, SpanEdge has not been evaluated against other approaches for geo-distributed stream processing, only against a centralized baseline, while the other papers have more extensive evaluations showing good

Paper	Programming Model Additions	Profiling Overhead	Solver Scalability	Evaluation Network?	Evaluation (Latency)	Evaluation (Bandwidth)
Optimal Operator Placement [1]	N/A	Low	Exponential (in MILP)	Emulated	100-150ms	Matches [33, 36, 44]
SpanEdge [2]	Local/global tasks	N/A	$\approx$ Linear (in program)	Emulated	$<$ 1ms local; $\approx$ 25ms global	$\approx \frac{1}{4}\times$ Storm
AWStream [3]	Maybe knobs for degradation	High	Exponential (in # knobs)	Real	$<$ 50ms; $\approx$ 100ms during low bandwidth	Adapts to available

Figure 11: Comparison of the three papers.

results against competing approaches. The evaluation in AWStream is the most convincing because it does not use a network emulator. All systems have sub-second latency, but SpanEdge was the only system to evaluate latency for local tasks (requiring no network communication). AWStream is the only system to adapt to network changes and to incorporate approximation (i.e., all other systems would fail in case the network is too constrained).

We have already mentioned some directions for future work specific to each system. More broadly, it would be desirable to conduct a thorough experimental comparison of these different approaches, and to see future approaches which perform well against all of them. Additional research could further illuminate important fundamental tradeoffs in this space: for instance, the tradeoff between solving overhead, additional programmer effort, and performance. Finally, some long-term challenges remain in processing geo-distributed streaming data: preserving data privacy; deploying complex geo-distributed applications, such as iterative computations and machine learning algorithms; and designing more effective and high-level programming abstractions with efficient implementations.

Primary References

[1] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS)*, pages 69–80. ACM, 2016.

[2] Hooman Peiro Sajjad, Ken Danniswara, Ahmad Al-Shishtawy, and Vladimir 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. IEEE, 2016.

[3] Ben Zhang, Xin Jin, Sylvia Ratnasamy, John Wawrzynek, and Edward A Lee. Awstream: Adaptive wide-area streaming analytics. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 236–252. ACM, 2018.

Other References

[4] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley Zdonik. The design of the Borealis stream processing engine. In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pages 277–289, 2005.

[5] Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[6] Yanif Ahmad and Uğur Çetintemel. Network-aware query processing for stream-based applications. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 456–467. VLDB Endowment, 2004.

[7] 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. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[8] 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.

[9] Boris Jan Bonfils and Philippe Bonnet. Adaptive and decentralized operator placement for in-network query processing. *Telecommunication Systems*, 26(2-4):389–409, 2004.

[10] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[11] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Optimal operator deployment and replication for elastic distributed data stream processing. *Concurrency and Computation: Practice and Experience*, 30(9):e4334, 2018.

[12] Valeria Cardellini, Francesco Lo Presti, Matteo Nardelli, and Gabriele Russo Russo. Decentralized self-adaptation for elastic data stream processing. *Future Generation Computer Systems*, 87:171–185, 2018.

[13] Bin Cheng, Apostolos Papageorgiou, Flavio Cirillo, and Ernoe Kovacs. Geelytics: Geo-distributed edge analytics for large scale iot systems based on dynamic topology. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 565–570. IEEE, 2015.

[14] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1 – 17, 2018.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[16] Raphael Eidenbenz and Thomas Locher. Task allocation for distributed stream processing. In *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9. IEEE, 2016.

[17] Apache Software Foundation. Apache flink. `https://flink.apache.org/`, 2019. [Online; accessed March 31, 2019].

[18] Apache Software Foundation. Apache samza. `http://samza.apache.org/`, 2019. [Online; accessed March 31, 2019].

[19] Apache Software Foundation. Apache spark streaming. `https://spark.apache.org/streaming/`, 2019. [Online; accessed March 31, 2019].

[20] Apache Software Foundation. Apache storm. `http://storm.apache.org/`, 2019. [Online; accessed March 31, 2019].

[21] Xinwei Fu, Talha Ghaffar, James C Davis, and Dongyoon Lee. Edgewise: a better stream processing engine for the edge. In *2019 USENIX Annual Technical Conference (USENIX ATC)*, pages 929–946, 2019.

[22] Lin Gu, Deze Zeng, Song Guo, Yong Xiang, and Jiankun Hu. A general communication cost optimization framework for big data stream processing in geo-distributed data centers. *IEEE Transactions on Computers*, 65(1):19–29, 2015.

[23] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Elastic stream processing for the internet of things. In *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, pages 100–107. IEEE, 2016.

[24] 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*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[25] Geetika T Lakshmanan, Ying Li, and Rob Strom. Placement strategies for internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, 2008.

[26] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. *ACM Comput. Surv.*, 43(3):19:1–19:51, April 2011.

[27] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 439–455, New York, NY, USA, 2013. ACM.

[28] Shadi A. Noghabi, Kartik Paramasivam, Yi Pan, Navina Ramesh, Jon Bringhurst, Indranil Gupta, and Roy H. Campbell. Samza: Stateful scalable stream processing at LinkedIn. *Proceedings of the VLDB Endowment*, 10(12):1634–1645, August 2017.

[29] Online. Aurora project page. `http://cs.brown.edu/research/aurora/`.

[30] Online. The borealis project. `http://cs.brown.edu/research/borealis/public/`.

[31] Online. Rust programming language. https://www.rust-lang.org/.

[32] Online. Storm github release commit 9d91adb. https://github.com/nathanmarz/storm/commit/9d91adbdbde22e91779b91eb40805f598da5b004.

[33] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49. IEEE, 2006.

[34] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and degradation in jetstream: Streaming analytics in the wide area. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 275–288, Berkeley, CA, USA, 2014. USENIX Association.

[35] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-driven stream processing at the edge. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 31–40. IEEE, 2017.

[36] Stamatia Rizou, Frank Durr, and Kurt Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *2010 Proceedings of 19th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2010.

[37] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[38] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*, pages 309–320. Elsevier, 2003.

[39] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, pages 159–170. VLDB Endowment, 2007.

[40] Twitter. Heron. https://apache.github.io/incubator-heron/, 2019. [Online; accessed March 31, 2019].

[41] Nikos Tziritas, Thanasis Loukopoulos, Samee U Khan, Cheng-Zhong Xu, and Albert Y Zomaya. On improving constrained single and group operator placement using evictions in big data environments. *IEEE Transactions on Services Computing*, 9(5):818–831, 2016.

[42] 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*, pages 374–389, 2017.

[43] Joel Wolf, Nikhil Bansal, Kirsten Hildrum, Sujay Parekh, Deepak Rajan, Rohit Wagle, Kun-Lung Wu, and Lisa Fleischer. Soda: An optimizing scheduler for large-scale stream-based distributed computer systems. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 306–325. Springer, 2008.

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

[45] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 423–438, New York, NY, USA, 2013. ACM.