Real-Time Batch Scheduling in Data-Parallel Complex Event Processing

Ruben Mayer, Muhammad Adnan Tariq, Kurt Rothermel
IPVS, University of Stuttgart
Stuttgart, Germany
first.last@ipvs.uni-stuttgart.de

ABSTRACT

Distributed Complex Event Processing has emerged as a well-established paradigm to detect situations of interest to an application from basic sensor streams, building an operator graph between sensors and applications. To enable operators to cope with high workload, the incoming data streams are split into—possibly overlapping—partitions which are processed in parallel by a set of operator instances. However, with increasing parallelization degree the network becomes a bottleneck, because events that are part of multiple different partitions are duplicated to multiple operator instances. In this paper, we address this problem and propose batch scheduling of overlapping partitions, i.e., assigning them to the same operator instance. Albeit reducing communication overhead, batch scheduling increases the processing latency of events – and thus inhibits the timely detection of situations – by inducing higher computational load on the operator instance. Controlling the trade-off between communication overhead and latency is challenging and cannot be solved with traditional reactive approaches. To this end, we propose an analytical batch scheduling controller building on prediction. Evaluations show that our approach is able to significantly save bandwidth and keep a latency bound in the operator instances.

1. INTRODUCTION

Modern applications need to be able to react to situations occurring in the surrounding world. Thus, a growing number of sensor streams need to be processed in order to detect situations which the application or user is interested in, e.g., the traffic situation in a smart city or the detection of a person in a video surveillance application. To detect situations from sensor streams, Distributed Complex Event Processing (DCEP) [16, 20, 4, 28] has been developed as a well-established paradigm building the bridge between sensors and consumers, i.e., applications or users that are interested in situations. A DCEP middleware deploys an operator graph in the network that incrementally detects patterns corresponding to situations in the sensor streams. In doing so, timeliness of pattern detection is of critical importance, as consumers need to react to occurring situations. This typically poses a soft latency bound on each operator of the DCEP system, because delayed situation detection leads to severe degradation of consumer benefits. For instance, late detection of a traffic jam leads to wrong routing decisions, and late detection of a person in a video surveillance application can mean that the relevant person has already left the scene when it is finally detected.

When running a DCEP system, high workload on the operators can lead to overload and long buffering delays when they process incoming streams only sequentially. To still keep a latency bound, the operator throughput must be increased by parallelization of event pattern detection. In this regard, data parallelization [19, 26] has been proposed as a powerful parallelization method. In a data parallelization framework, incoming event streams of an operator are split into partitions that can be processed in parallel by an arbitrary number of operator instances, e.g., deployed in a cloud data center. To ensure consistency, each partition comprises all events needed in order to detect a pattern. This means that different partitions can overlap, i.e., events are part of multiple partitions [26, 22].

When splitting incoming event streams, the data parallelization framework assigns a partition to an operator instance when the start of the partition is detected. In doing so, assigning overlapping partitions to different operator instances results in increasing network load, as events that are part of multiple different partitions are duplicated to multiple operator instances. In the worst case, an event may be transmitted to all operator instances. With increasing parallelization degree, the network can become the new bottleneck in the middleware that is limiting the scalability. Especially, network-intensive applications have been identified as a major cause of network congestions in cloud data centers [23, 7, 18]. To cope with the ever-increasing workload, this bottleneck must be overcome.

A way to tackle the network bottleneck is batch scheduling of subsequent overlapping partitions, i.e., assigning them to the same operator instance [17]. That way, events from the overlap only need to be transferred once. However, at the same time, the operator instance must process more partitions in a shorter time. This can lead to temporary overload, so that events get buffered and queuing latency is accumulating. Nevertheless, latency between arrival of an event and its successful processing must not exceed a given latency bound. We address the following challenges in batching the

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

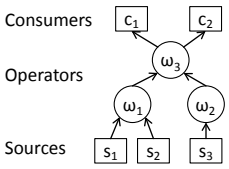


Figure 1: DCEP operator graph.

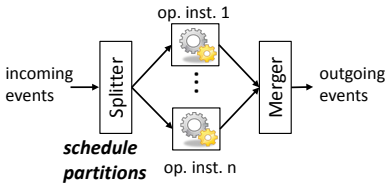


Figure 2: Data parallelization framework.

optimal amount of partitions, which cannot be solved with state-of-the-art scheduling algorithms from stream processing [10, 25, 6].

- **Per-event latency**: Each incoming event at an operator can potentially trigger the detection of a pattern leading to a situation detection. Therefore, a latency bound should be kept for *each single event*.
- **Partition overlap**: The overlap between partitions of a batch influences the processing load induced by each event, because each event has to be processed in the context of each partition it is part of. Moreover, the scheduling decision is made on open partitions, i.e., the events and the overlap of a partition are not known at scheduling time.
- **Automatic adaptation**: A batch scheduling controller should be able to automatically adapt to changing workload conditions without being manually trained for those conditions beforehand.

Toward this end, we make the following contributions in this paper. (1) Based on evaluations from different DCEP operators, we identify key factors that influence the latency in operator instances. In particular, we identify factors that have not been regarded in related work before. (2) Taking into account the identified key factors, we propose a model-based batch scheduling controller. The model allows to predict the latency induced in operator instances when assigning partitions. (3) We provide extensive evaluations of the system behavior in two different scenarios, showing that our approach minimizes communication overhead while operator instances keep a required latency bound even when the system faces heavily fluctuating workloads.

2. DATA-PARALLEL DCEP SYSTEMS

Before introducing the methods for bandwidth-efficient batch scheduling, we introduce a model of a data-parallel DCEP system, as proposed in recent work [19, 26]. We aim to develop a batch scheduling method that is suitable for a large number of such systems.

A DCEP system builds an operator graph interconnecting event sources, operators and consumers by event streams. For example, Figure 1 depicts a DCEP deployment with 3 sources, 3 operators and 2 consumers. An event e consists of its payload and a header containing its event type. Events from all streams inherently have a well-defined total order[1]. When receiving events from different incoming streams, operators assign sequence numbers to the events according to the global order, process the events in-order and emit outgoing events to their successors in the operator graph. In doing so, operators detect event patterns in finite, non-empty subsets of their incoming event streams—called *partitions* and denoted by π [22].

To cope with high workload, each operator is executed in a data parallelization framework (cf. Figure 2). It consists of a split–process–merge architecture [19, 26]. A splitter divides the incoming event streams of the operator into partitions. To ensure consistency, each partition comprises all events needed in order to detect a pattern. Such patterns can be defined in event specification languages like Snoop [12] or Tesla [15], and comprise, e.g., sequence patterns, window-based patterns, aperiodic patterns, and others. Those partitions are then scheduled (i.e., assigned) to an elastic set of operator instances which simultaneously process their assigned partitions. Finally, a merger orders the events emitted by the operator instances into a deterministic sequence.

To allow for a virtually unlimited parallelization degree, all components are deployed on (possibly virtual) distinct shared-nothing hosts, and each of them can access a dedicated set of resources in terms of CPU and memory, i.e., we do not require shared memory between different operator instances or between the splitter and the operator instances. The hosts of the components are inter-connected by unicast communication channels that guarantee eventual in-order delivery of streamed events. Focusing on the main technical challenges in this paper, we constrain ourselves to homogeneous hosts to deploy operator instances.

According to the pattern specification and the occurring events, partitions can have different sizes and a different number of events can occur between two start events of subsequent partitions. We denote the period of time that a partition spans, i.e., the time between the first event and the last event of a partition, as the *partition scope*, ps. Further, we denote the period of time between two start events of subsequent partitions as the *partition shift*, Δ. Splitting the incoming event streams consistently can, for instance, be achieved by evaluating user-specified logical predicates [26] that signal which events are needed in order to detect a queried pattern. When the start of a new partition has been detected, this partition is assigned to an operator instance according to a scheduling algorithm. In an operator instance, incoming events are processed sequentially. Within the scope of each partition, an event has a different context with respect to its processing. Therefore, when processing an event e, the operator instance sequentially processes e in the scope of each partition that e is part of.

Example: In the scenario in Figure 3, the pattern to be detected is "within one minute after occurrence of an event of type A, a sequence of events of type B and C occurs"—i.e., $Aperiodic[A; Sequence(B; C);\ A.timestamp\ +1min]$ in Snoop syntax[12]. The splitter opens a partition whenever an event of type A occurs, and closes the partition after one minute. The operator instances check whether in a partition, events of type B and C occur in the right order. Two overlapping partitions π_x and π_y have been assigned to the same operator instance i. When i processes an event, e.g., C_1, this event has a different context in π_x than in π_y: In π_x, the sequence (B;C) is detected, while in π_y, the sequence is not detected. In checking the occurrences of the sequence pattern in different partitions, operator instance i processes C_1 sequentially first in π_x and then in π_y.

From the event consumer's point of view, the situation detection latency is the period from the occurrence of a source

[1]This order can, for instance, be established on time-stamps assigned by event sources with synchronized clocks, so that it reflects the ordering of physical occurrence of source events.

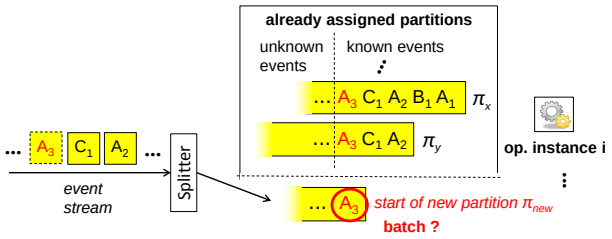


Figure 3: Batch scheduling decision.

event that signals a situation of interest until the situation is actually detected and signaled to the consumer. As the delayed detection of a situation degrades the benefits for the application, it poses a soft latency bound on the overall situation detection. That means, that violations of the latency bound shall, if possible, be avoided. The situation detection latency spans the whole operator graph of the DCEP middleware and is sub-divided into latency budgets for each operator in the operator graph. In each operator, the splitter and the merger induce latency for splitting the streams into partitions and merging the results. Because scheduling partitions to operator instances significantly influences the latency induced in each operator instance, in this paper, we focus on batch scheduling suitable amounts of partitions to operator instances such that a latency bound in those operator instances is kept.

We define the *operational latency* of e, $\lambda_o(e)$, as the period between the point in time when e arrives at an operator instance and the point in time when e is completely processed in all assigned partitions in this operator instance. When, at the time of arrival of e, the operator instance is still busy with processing earlier events, e waits in a queue until its processing can start. This is called *queuing latency* of e, $\lambda_q(e)$. Then, e is processed, which induces the *processing latency* of e, $\lambda_p(e)$, the time from starting to process e until e is processed in all assigned partitions. Overall, the operational latency of an event is a combination of its queuing latency and processing latency, i.e., $\lambda_o(e) = \lambda_q(e) + \lambda_p(e)$.

To explain the trade-off between operational latency and communication overhead in batch scheduling, refer again to the example in Figure 3. An event A_3 arrives at the splitter and the splitter detects that A_3 starts a new partition π_{new} which now has to be scheduled to one of the available operator instances. Suppose a set of previous partitions Π_{old} =(..., π_x, π_y) had already been scheduled to a specific operator instance i. Events before A_3 in Π_{old} have been transferred to operator instance i. However, further events arriving after A_3 can as well be part of some of the partitions in Π_{old}; hence, they are transferred to operator instance i, too. When scheduling π_{new} to operator instance i, communication overhead can be reduced, because events overlapping between π_{new} and Π_{old} do not need to be transferred to multiple different operator instances. On the other hand, they need to be processed additionally in the scope of π_{new}, inducing higher processing latency. The splitter has to decide whether π_{new} can be assigned to operator instance i such that the operational latency does not increase beyond a given latency bound.

Next, we analyze batch scheduling in more detail.

3. BATCH SCHEDULING

The batch scheduling controller has to batch as many partitions as possible to the same operator instance such that the operational latency of events in that instance will not exceed a latency bound LB. This is noted as the *batch scheduling problem* in data-parallel DCEP operators. To solve the problem, in this section, we make the following contributions. First, in Section 3.1, we identify and thoroughly analyze *key factors* that influence the operational latency in an operator instance. We conclude that the impact of key factors on operational latency in an operator instance is complex and depends on the workload as well as on the operator. Further, in Section 3.2, we show the difficulties we encountered when developing a reactive batch scheduling controller that works without a latency model.

3.1 Key Factors

In the following, we first identify and analyze key factors that influence the processing latency of events in the scope of a *single* partition. Based on that, we identify and analyze key factors that influence operational latency in a whole *batch* of partitions. To this end, we evaluate two different DCEP operators: a traffic monitoring and a face recognition operator. We ran all experiments on the computer cluster described in Section 5 with a parallelization degree of 8.

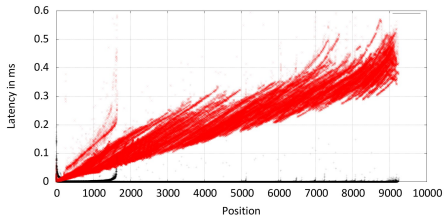
Traffic monitoring operator. A traffic monitoring application is interested in violations of an overtaking ban, so that the transgressor can be warned or punished. To this end, two cameras at two different locations ($L1$ and $L2$) on a highway capture video streams of vehicles passing by. To detect overtaking maneuvers, a traffic monitoring operator ω is deployed between the cameras and the application. When a vehicle passes a camera, an event is emitted to ω, containing a time-stamp, the type (location $L1$ or $L2$), and the number plate. To detect the violations, ω uses an *aperiodic* partition window: Whenever a vehicle a passes $L1$, a partition π is opened, and when the same vehicle passes $L2$, π is closed. Another vehicle b that appears in the $L1$ stream within π has passed $L1$ after a. When b appears again in π in the $L2$ stream, it has passed $L2$ before a. If this is the case, b has overtaken a and thus violated the traffic rules. The query in ω can be expressed in CEP query languages, e.g., in Snoop [12] language as an aperiodic operator: Aperiodic(A; B; C) with A $\rightarrow$ ⟨plate=a, type=$L1$⟩, B $\rightarrow$ Sequence(⟨plate=b, type=$L1$⟩;⟨plate=b, type=$L2$⟩), C $\rightarrow$ ⟨plate=a; type=$L2$⟩[2].

Face recognition operator. A face recognition application wants to know whether a person of interest is currently located in a specific area. To this end, pictures of detected faces from a camera are transferred to a face recognition operator ω. Further, query events from users querying whether a certain person is in the current video stream are sent to ω, containing a set of pictures of the person and a time frame within which the person shall be detected. ω uses a face recognition algorithm in order to detect whether the queried person is in the stream. This query can be resembled by an aperiodic operator Aperiodic(A; B; C) with A $\rightarrow$ ⟨type=*query*, time=t⟩, B $\rightarrow$ ⟨type=*face*, "`face_match(A)`"⟩, C $\rightarrow$ *time* $\geq t +$ *time frame*.

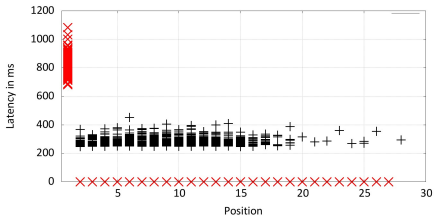
3.1.1 Processing Latency of Events in a Partition

When processing a *single* partition in an operator instance, each event imposes a specific processing latency. This is different from stream processing operators where the pro-

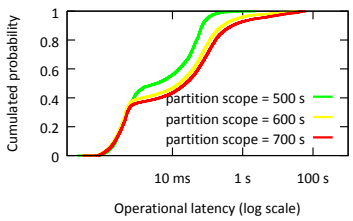
[2]Aperiodic(A; B; C): Between the occurrence of two (complex) events A and C, the (complex) event B occurs.



(a) Traffic monitoring operator: Processing latency of events in different positions in a partition. $L1$ events: black, $L2$ events: red.



(b) Face recognition operator: Processing latency of events in different positions in a partition. Face events: black, query events: red.



(c) Traffic monitoring operator: Operational latency at $LB = 1s$ and different partition scopes with current-reactive batch scheduling at $TH = 10\%$.

Figure 4: Evaluations.

cessing latency of an event in a partition is considered fixed [6, 32]. We identified two key factors that influence the processing latency of an event in a partition: its position and its type.

Position of Event. When processing events of a partition, internal state is gathered in an operator [11, 5], which can influence the processing latency of events. For instance, in the traffic monitoring operator, an *L2* event e_{L2} can potentially complete a pattern Sequence(⟨plate=b, type=$L1$⟩;⟨plate=b, type=$L2$⟩) or close the partition. Therefore, e_{L2} is compared to all *L1* events that have been seen in the partition before. Thus, with a higher position of e_{L2}, its processing latency increases, as evaluated in Figure 4a. However, the processing latency of events does not necessarily increase with position. In the face recognition operator, each face event is compared to the first query event of the partition; the `face_match` function imposes the same processing latency in each event position (cf. Figure 4b).

Event type. Event types are a fundamental concept in DCEP. To the best of our knowledge, all complex event query languages, for example Snoop [12], Amit [1], SASE [30] and Tesla [15], allow for the definition of event patterns based on event types—e.g. SEQ(A;B), a sequence of events of type A and B. In the traffic monitoring operator, different event types are processed in a different way. *L1* events are simply added to a list of seen events, while *L2* events are compared to the seen events (cf. Figure 4a). In the face recognition operator, *query events* are processed by building a face model of the queried person, while *face events* are processed by comparing them to the established face model of the partition (cf. Figure 4b). In both operators, we see different processing latencies depending on the event types.

3.1.2 Operational Latency in a Batch of Partitions

In a *batch* of partitions, different partitions may overlap. When the batch scheduling controller assigns a partition to an operator instance that overlaps with already assigned partitions, the processing latency of all events in the overlap is influenced, as events are processed sequentially in the scope of their partitions. Recall that a partition has to comprise all events needed in order to detect a queried pattern. Therefore, the overlap of different partitions cannot be changed by the batch scheduling controller. That is different from batch scheduling problems handled in stream processing, where batches are considered to be arbitrarily large, *non-overlapping* sets of events, and batch scheduling decides how many *events* shall be batched to a processing node [17, 25].

In the following, we identify key factors influencing the overlap of partitions and analyze their impact on operational latency in operator instances. To this end, we run experiments with the traffic monitoring operator and the face recognition operator. In each experiment, simulating different traffic densities and different numbers of persons in a video frame, one key factor value is changed while all other key factors are kept constant, and the differences in operational latency peaks are analyzed (cf. Figure 5). For each experiment, more than 370,000 operational latency measurements have been taken.

Batch size. The batch size, i.e., the number of partitions assigned to one operator instance in a batch, influences the overlap of the partitions, and hence, the operational latency of events. This seems to be a trivial insight. However, the relation between batch size and operational latency peak is not trivial. In the traffic monitoring operator, increasing the batch size by 50 % and then by further 33 % induces an increase in operational latency peak by 317 % and 15 %, respectively (cf. Figure 5a, experiments #3, #4 and #5). In the face recognition operator, the relation between batch size and operational latency seems to be proportional (cf. Figure 5b).

Inter-arrival time (iat). Given a fixed batch size, the inter-arrival time *iat* of events influences the queuing latency of events. Further, it can influence the number of events in the partitions, e.g., in time-based partitions. The number of events in partitions influences their overlap, which, in turn, influences the processing latency of the events. Thus, there is a complex relation between *iat* and operational latency. In the traffic monitoring operator, we decreased the average *iat* of events first by 17 %, and then by further 20 %. This induced an increase in operational latency peak by 54 % and 551 %, respectively (cf. Figure 5a, experiments #1, #2 and #3). Similarly, in the face recognition operator, decreasing the average *iat* of events first by 40 % and then by further 28.5 %, led to an increase in operational latency peak by 81 % and 45 %, respectively (cf. Figure 5b).

Partition scope (ps). The partition scope *ps*—i.e., the time between the start and end event of a partition—depends on the queried patterns to be detected by the DCEP operator. It can be fixed to a specific time, e.g., when the query depends on a time-based window [2], but it can also depend on the occurrence of specific events, e.g., in aperiodic queries or queries that define a sequence of specific events [15, 12]. For instance, in the traffic monitoring operator, the start and end of a partition depend on the speed of the vehicles, as a partition starts when a vehicle passes *L1* and ends when the same vehicle passes *L2*. When the speed of a vehicle is lower, the time spanned by the partition opened from this vehicle is larger. Therefore, the size and overlap of

scenario parameters				measurements			
#	batch size	avg. iat (s)	ps (s)	max. op. lat. (s)	feedback delay (s)	max. q. length	feedback delay (s)
1	500	0.15	900	2.4	725.5	15	773.6
2	500	0.125	900	3.7	757.7	27	724.8
3	500	0.1	900	24.1	699.0	248	810.2
4	750	0.1	900	100.6	800.8	1029	844.0
5	1,000	0.1	900	116.1	824.3	1194	795.6
6	1,000	0.1	1000	197.8	1,041.8	1699	999.0
7	1,000	0.1	1100	199.2	1,179.2	1898	1,100.0

(a) Traffic monitoring operator.

scenario parameters				measurements			
#	batch size	avg. iat (s)	ps (s)	max. op. lat. (s)	feedback delay (s)	max. q. length	feedback delay (s)
1	10	0.667	10	37.9	46.3	43	10.1
2	10	0.4	10	68.7	77.1	84	9.4
3	10	0.286	10	99.7	108.0	115	10.6
4	15	0.286	10	145.1	153.2	164	8.3
5	20	0.286	10	195.1	200.7	191	10.2
6	20	0.286	15	289.4	301.8	234	14.4
7	20	0.286	20	392.1	410.1	258	19.6

(b) Face recognition operator.

Figure 5: Max. operational latency, queue length and feedback delays.

partitions can change even when the batch size and *iat* stay the same. This is different from stream processing, where only partitions of fixed size and fixed slide— time- or count-based—are analyzed [6]. In the traffic monitoring operator, we increased *ps* in the traffic monitoring operator by 11 %, and then by further 10 %. This induced an increase in operational latency peak by 70 % and 1 %, respectively (Figure 5a, experiments #5, #6 and #7). In the face recognition operator, however, increasing *ps* led to a proportional increase in operational latency peaks.

From the observations on key factors that influence operational latency when processing a batch of partitions, we see that building a direct mapping from *batch size*, *inter-arrival time* and *partition scope* to operational latency peaks in operator instances is hard. The relation between key factors and operational latency peaks that occur in operator instances is complex, and different in different operators. An off-line trained model, hence, does not suffice; due to the complex relations between key factors, it is hard to train a model that can make reliable predictions outside of the learned parameter value ranges. Further, domain knowledge alone is not enough in order to hand-craft a latency model: Knowledge about the operator implementation does not necessarily help in understanding the relations between the identified key factors and the operational latency peak.

In the following, we discuss whether the need for a latency model predicting the operational latency can be completely avoided by employing a reactive batch scheduling controller.

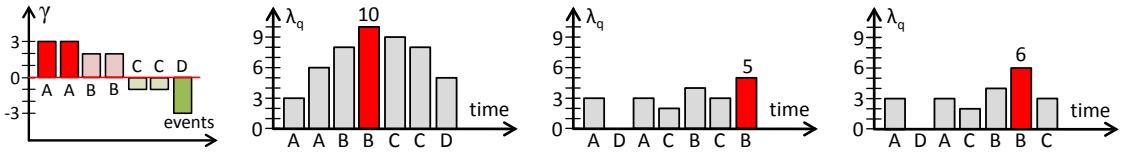
3.2 Reactive Controllers

Here, we discuss the difficulties involved in devising a reactive batch scheduling controller. Reactive controllers are widely used in scheduling algorithms in the related field of parallel stream processing systems [17, 25]. The basic idea of a reactive controller is that it schedules partitions according to *feedback parameters* (like operational latency or queue length) from the operator instances that indicate how many partitions can be batched. In the following, we point out the differences in batch scheduling in data-parallel DCEP operators to scheduling problems that have been solved with reactive controllers. Then, we analyze operational latency and queue length of operator instances in the scope of the scenarios described in Section 3.1 in detail and show that none of these parameters provides reliable feedback to implement a reactive controller.

In data-parallel DCEP operators, in order to maintain the latency bound for each event, the batch scheduling controller decides at the *start* of a partition to which instance this partition is scheduled. Then, it directs all events that arrive in the scope of that partition to the corresponding instance. It is infeasible for the controller to wait until all events of the partition are present and then schedule the partition; it would take too much time in view of per-event latency bounds. After assigning a partition to an operator instance at the occurrence of its start event, many other events of that partition arrive until the partition is finally closed. Thus, over the *whole time span* of the partition, feedback parameters in the operator instance are influenced by the scheduling decision, i.e., a long time after the scheduling decision has been made. That poses a completely different problem from other batch scheduling problems that are tackled with reactive batch scheduling, e.g., the problem of scheduling batches of events in streamed batch processing [17], where a controller *first* builds a batch of available events and *then* assigns it to an operator instance.

Therefore, in data-parallel DCEP operators, there can be a high delay between assigning a partition to an operator instance and the occurrence of the peak value of the feedback parameters in that operator instance. In the following, we denote this delay as the *feedback delay*. In Figure 5a, we have measured the feedback delay of operational latency and of queue length in the different runs of the traffic monitoring operator under different conditions; a feedback delay of 699 to 1,179 *seconds* occurred for both parameters. In that time, many subsequent batch scheduling decisions have to be made by the controller. At the same time, key factors like inter-arrival time, partition scope, event types, etc. continuously change. An additional complication is that the feedback delay is not constant, so that the controller cannot rely on it; it is not clear whether the parameter measured in an operator instance is already the peak value or how much further it will grow.

To mitigate high feedback delays, we tried to measure the *current* operational latency in operator instances to deduce some early *trend* and then decide whether to batch more partitions to an operator instance based on that trend. This resulted in a *current-reactive* controller: Partitions are batched to the same operator instance until at the instance, a percentage threshold TH of the operational latency bound LB is reached; subsequent partitions are scheduled to the next operator instance. This, however, poses the question how to set TH. An experiment shows that a static TH is not enough to keep the latency bound. We run evaluations using the traffic monitoring operator at an average inter-arrival time of cars of 200 ms, aiming to keep $LB = 1s$. With $TH = 10\%$, reactive batch scheduling more or less was able to keep LB when *ps* was not higher than 500 s and the average *iat* was not lower than 200 ms (cf. Figure 4c). However, at a *ps* of 600 s and 700 s, $TH = 10\%$ led to systematically wrong batch scheduling decisions; LB was violated by a factor of almost 100. Obviously, TH has to be adapted to the changing key factor values. In doing so, the feedback



(a) Latency gains. (b) W-C: Latency peak. (c) B-C: Latency peak. (d) M-C: Latency peak.

Figure 6: Different sequences of negative and positive gains.

to change *TH* is available only after *LB* already has been violated, i.e., after a long feedback delay. The same problems apply when using the queue length peaks as a feedback parameter in the traffic monitoring operator: The feedback delay is high. Same as with operational latency, using the current queue length needs a suitable threshold, which in turn has to be adapted to changing key factor values.

In the face recognition operator, partition scopes are much smaller. While the feedback delay of operational latency peaks is still high (46 to 410 *seconds*), the feedback delay of the queue length peaks is smaller (8 to 20 seconds; cf. Figure 5b). However, this does not automatically make the queue length peaks a good parameter for reactive controllers. First of all, 20 seconds is still a long time; in the real-world workloads analyzed in Section 5, sudden bursts demand for an even faster reaction. Second, the relation between queue length peak and operational latency peak is not trivial; the operational latency peak does not necessarily occur when the most events are in the queue, but rather when the most expensive events are in the queue. This demands for a more thorough analysis. We conclude that neither operational latency nor queue length are a reliable feedback parameter for a purely reactive batch scheduling controller.

Instead of pure feedback mechanisms, which are too simplistic to solve the problem, our approach uses a simple, yet powerful latency model. It takes into account feedback from operator instances, but also includes a prediction and analysis step.

4. MODEL-BASED CONTROLLER

To solve the batch scheduling problem, the controller has to predict whether the operational latency peak will be higher than *LB* when batching a new partition π_{new}. To make this prediction, in this chapter, we introduce a latency model. We aim to find the right balance between the complexity needed in order to make good predictions of the operational latency peak, the reasonable consideration of feedback from operator instances and of domain expert knowledge, and the accuracy and precision of the model.

4.1 Basic Approach

Recall that the operational latency of an event e is built up of its queuing and processing latency: $\lambda_o(e) = \lambda_q(e) + \lambda_p(e)$. If the processing latency $\lambda_p(e)$ of an event is higher than the inter-arrival time *iat* to its successor event, this imposes additional queuing latency to the successor event. On the other hand, if $\lambda_p(e)$ is smaller than *iat*, the queuing latency of the successor event becomes smaller or even zero, i.e., e does not induce queuing latency for the successor event. In the following, we refer to the difference between λ_p and *iat* as the *gain* γ of an event: $\gamma(e) = \lambda_p(e) - iat$. If $\lambda_p(e) > iat$, we speak of a *negative gain*; else, we speak of a *positive gain*[3]. In Figure 6a, we provide an example. Suppose that the *iat* between events is 5 time units (TU), and the partition contains 7 events: 2 events of type A impose $\lambda_p = 8$ TU, 2 events of type B impose $\lambda_p = 7$ TU, 2 events of type C impose $\lambda_p = 4$ TU, and 1 event of type D imposes $\lambda_p = 2$ TU. Then, the gains of the single events are, accordingly, between +3 and -3 TU (+3 for type A, +2 for B, -1 for C, -3 for D).

Now, for the overall partition π_{new}, the aggregated gains of the set of events with $\lambda_p(e) > iat$ are termed as the *total negative gain*: $\Gamma^- = \sum \gamma(e) : e \in \pi_{new} \wedge \lambda_p(e) > iat$. In the given example (Figure 6a), those are the events of type A and B; hence, $\Gamma^- = 3+3+2+2 = 10$ TU. The aggregated gains of the set of events with $\lambda_p(e) < iat$ are termed as the *total positive gain*[4]: $\Gamma^+ = \sum \gamma(e) : e \in \pi_{new} \wedge \lambda_p(e) < iat$. In the given example (Figure 6a), those are the events of type C and D; hence, $\Gamma^+ = (-1) + (-1) + (-3) = -5$ TU.

After defining the total negative and positive gains, in the following, we analyze possible sequences of negative and positive gains and the impact on the queuing latency peak λ_q^{max}. In Figure 6b, first all negative gains occur, followed by all positive gains. This is the worst case with respect to λ_q^{max}; in the example sequence, $\lambda_q^{max} = 10$ TU. Note, that also any other sequence of events of types A and B would lead to the same λ_q^{max}. In the worst case, hence, $\lambda_q^{max} = \Gamma^-$. However, an interleaving between negative and positive gains is possible as well. Take a look at Figures 6c and 6d: In the examples, the events with negative and positive gains interleave to a different extent. This leads to different values of λ_q^{max}, because although the queuing latency is increased by events with negative gains, events with positive gains compensate for that; a successor event of an event with positive gain faces a lower queuing latency.

The actual sequence of events with negative and positive gains in π_{new} is very difficult to predict. It would essentially correspond to predicting each single event in π_{new} and its *iat*. To account for the discussed interleaving of events with negative and positive gains, therefore, we introduce a *compensation factor* α. α allows for modeling the extent of interleaving of negative and positive gains without the need to explicitly define the sequence of events in π_{new} in the prediction: $\lambda_q^{max} = \Gamma^- + \alpha * \Gamma^+$. Taking a look at the best-case example in Figure 6c, we see that the negative and positive gains are maximally interleaving, hence, $\alpha = 1$. Accordingly, $\lambda_q^{max} = 10 + 1 * (-5) = 5$. Figure 6d exemplifies an event sequence in between the worst- and best-case: Parts of the positive gains are interleaving with the negative gains, hence, $\alpha = 0.8$. Accordingly, $\lambda_q^{max} = 10 + 0.8 * (-5) = 6$.

Please notice, that the first event of π_{new} might already face a queuing latency λ_q^{init} at its arrival. This can be due to previous partitions that had been scheduled to the same operator instance. Hence, the final formula to calculate the

[3]Negative gains are positive numbers and positive gains are negative numbers. The terminology refers to the impact of an event on the feasibility to schedule a partition in a batch.

[4]If $\lambda_p(e) = iat$, neither negative nor positive gains occur.

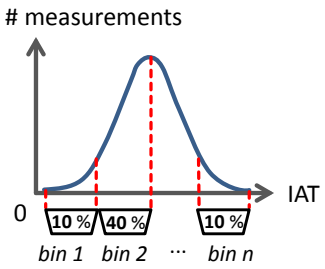


Figure 7: IAT Bins.

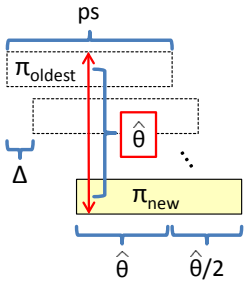


Figure 8: Overlap.

queuing latency peak is:[5]

$$\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+, \alpha \in [0,1].$$

From the queuing latency peak λ_q^{max}, the operational latency peak λ_o^{max} is calculated using the maximal processing latency λ_p^{max} of any event in π_{new}. This bases on the pessimistic assumption that the most expensive event occurs right at the queuing latency peak; as we do not know the event sequence, this assumption is justified by the goal to avoid underestimations of λ_o^{max}. Hence,

$$\lambda_o^{max} = \lambda_q^{max} + \lambda_p^{max}.$$

Using this latency model, the operational latency peak can be predicted, and the scheduling decision—to batch or not to batch—can be made accordingly. In the following sub-section, we describe how the parameters of the model are predicted.

4.2 Prediction of Model Parameters

The proposed latency model builds on predicting the total sum of negative and positive gains of all events in π_{new}; i.e., it does not regard individual events, but it regards events in π_{new} as *sets* of events imposing negative or positive gains. Hence, it builds on the prediction of the *set* of events in π_{new}, including their processing latency λ_p and their inter-arrival time *iat*. Further, a prediction of the initial queuing latency λ_q^{init} and the compensation factor α is needed. Based on those values, the model predicts the operational latency peak. In this section, we discuss appropriate prediction methods and algorithms.

Inter-arrival time. To model the inter-arrival time *iat* of events, we propose four possible methods with different accuracy and complexity:

1. Using the mean value $\overline{iat}'$ measured in the splitter in the past *mtime* time units.
2. Using $\overline{iat}'$ minus a negative bias. This way, the model can account for changes in the average *iat* between the monitored value $\overline{iat}'$ and the value that will occur in π_{new}. The negative bias is modeled based on a factor δ_{iat} of standard deviations σ of the monitored *iat*s, e.g., 1 standard deviation or 2 standard deviations. Then, $iat = \overline{iat}' - \delta_{iat} * \sigma$.
3. To increase the precision of the model, instead of the overall mean *iat*, the splitter monitors the current distribution of inter-arrival times in a discrete model (cf. Figure 7). In the model, the range of measured *iat* values is divided into a number of equally-sized *bins*. The measured *iat*s are sorted into the corresponding bin; for each bin B_i, the mean value $\overline{iat(B_i)}'$ and the weight of the bin $weight(B_i)$—i.e. ratio of number of entries in the bin to total number of measurements in all bins—is computed. The number of bins manages the accuracy of the model.
4. A combination of bins and negative bias. In each bin B_i, we predict $iat(B_i) = \overline{iat(B_i)}' - \delta_{iat} * \sigma$.

In our evaluation section, we discuss the pros and cons of the different methods and also give hints on how to set-up the parameters.

Processing Latency. In our model, λ_p depends on the overlap Θ and the processing latency in a single partition λ_p^π: $\lambda_p = \Theta * \lambda_p^\pi$. As discussed in Section 3, λ_p^π depends on the event type and the position in a partition. Hence, first of all, our model differentiates between different event types. This design decision has two consequences: First, the prediction model of λ_p^π takes into account the type, i.e., predict $\lambda_p^\pi(type)$, the in-partition processing latency of events of a specific type. Second, the set of events in π_{new} is predicted with respect to the number of events of different types.

For predicting $\lambda_p^\pi(type)$, we propose the same methods as for predicting *iat*: Using the mean value $\overline{\lambda_p^\pi}'(type)$ measured in a monitoring window, using the mean plus a negative bias of δ_{λ_p} standard deviations, using *latency bins* to resemble the cdf of processing latencies, or using a combination of bins and negative bias. Same as in *iat* bins, in each latency bin B_l, we predict $\lambda_p^\pi(B_l) = \overline{\lambda_p^\pi(B_i)}' + \delta_{\lambda_p} * \sigma$, i.e., the measured mean in-partition processing latency in the latency bin plus a factor δ_{λ_p} of standard deviations. In our evaluation section, we discuss the pros and cons of the modeling strategies and also give hints on how to set the parameters. The advantage of monitoring the current (distribution of) $\lambda_p^\pi(type)$ in the operator instances over building a position-dependent latency model is that we can *implicitly* incorporate the *position-dependency*: When the (distribution of) positions of events in partitions change, e.g., due to changing workload or changing partition scopes, this is reflected in the monitored current (distribution of) $\lambda_p^\pi(type)$ values. We do not need to explicitly model the positions of individual events.

The overlap Θ for all events of π_{new} is modeled as the average overlap of events of π_{new} in the current batch, denoted by $\overline{\Theta}$. Predicting $\overline{\Theta}$ is performed according to the following model (cf. Figure 8). When π_{new} is scheduled in a batch of already opened partitions, a number of events in π_{new} has the current overlap $\hat{\Theta}$, until the oldest open partition π_{oldest} in the batch closes. From closing π_{oldest} until closing π_{new}, the overlap decreases step-wise in regular intervals, each time a partition between π_{oldest} and π_{new} is closed. In that phase, the average overlap is $\hat{\Theta}/2$. In order to compute $\overline{\Theta}$, we weigh the ratio of events with overlap $\hat{\Theta}$ to the events with overlap $\hat{\Theta}/2$. In doing so, we assume in our model that all partitions in the batch have the same partition scope *ps*, and between the start of two partitions, there is the same shift Δ; *ps* and Δ are measured in the splitter at regular intervals to keep them up to date at each scheduling decision.

At the start of π_{new}, π_{oldest} is already open since $(\hat{\Theta}-1)*\Delta$ time units, as $\hat{\Theta}-1$ is the number of partitions between π_{oldest} and π_{new} that were opened in intervals of Δ time units. Therefore, π_{oldest} stays open for $ps - (\hat{\Theta}-1) * \Delta$ more time units. When π_{oldest} closes, the phase of closing partitions starts, spanning $(\hat{\Theta}-1) * \Delta$ time units. Hence, the weighed average overlap is computed as follows:

$$\overline{\Theta} = \frac{(ps-(\hat{\Theta}-1)*\Delta)*\hat{\Theta}+(\hat{\Theta}-1)*\Delta*\hat{\Theta}/2}{ps}.$$

[5]For the sake of readability, we did not mention in the text that $\lambda_q^{max} = \lambda_q^{init}$, if $\Gamma^- + \alpha * \Gamma^+ < 0$.

Set of Events. For predicting the set of events in π_{new}, there are three significant factors in the model: (1) The partition scope ps, (2) the (average) iat, and (3) the ratio of different event types, denoted as $ratio(type)$, that models which percentage of events in π_{new} is of a specific $type$. These factors are gained from monitoring them in the incoming event stream in the splitter in the past $mtime$ time units. As discussed earlier, a negative bias of δ_{iat} standard deviations $\sigma(iat)$ of the monitored iat can be considered in the prediction, such that $iat = \overline{iat}' - \delta_{iat} * \sigma(iat)$. Then, the total number of events n is predicted as $n = \frac{ps}{iat}$, and the number of events of a specific $type$, denoted by $\#(type)$, is predicted as $ratio(type) * n$.

Initial Queuing Latency. The initial queuing latency is predicted for each operator instance separately, depending on the content of the incoming event queue. To this end, operator instances report the number of events of each type and their average overlap $\overline{\Theta}$ in the assigned partitions in regular intervals to the splitter. Whenever making a batch scheduling decision, the splitter calculates λ_q^{init} of an operator instance as the sum of the processing latencies of all reported events in its queue: $\lambda_q^{init} = \sum_{types} \#events * \overline{\Theta} * \lambda_p^{\pi}(type)$.

Compensation Factor. For modeling the compensation factor α, there are two possibilities.

First, we propose a heuristic, denoted as T-COUNT, for adapting α based on the current extent of interleaving between events with different processing latency in the incoming stream. To this end, events are divided into two groups, based on their in-partition processing latency λ_p^{π}: the group of events with higher λ_p^{π} is denoted by T^- and the group of events with lower λ_p^{π} is denoted by T^+. The distinction between the groups is made based on the average $\lambda_p^{\pi}(type)$ of the event types; there is one half of event types that has higher $\lambda_p^{\pi}(type)$ than the other half of event types. Events of any of the types that pose higher processing latencies are grouped into T^-, other events are grouped into T^+. The splitter continuously counts in a monitoring window of temporal size $mtime$, how many events in T^-, denoted by c^-, and how many events in T^+, denoted by c^+, occur. Further, the splitter counts how often events in T^- and T^+ follow each other, i.e., the number of transitions, denoted by c^t. The maximal number of transitions is $2 * min\{c^+, c^-\}$. Trivially, the minimum number of transitions is 1. Then, α is predicted as the proportion of c^t to the maximal number of transitions: $\alpha = \frac{c^t - 1}{2 * min\{c^+, c^-\}}$.

Second, a domain expert can also set a fixed or dynamic value of α based on off-line training, if the characteristics of the expected workloads are known beforehand.

4.3 Scheduling Algorithm

Having a prediction of the set of events in π_{new}, processing latencies and inter-arrival times, the batch scheduling controller predicts the total negative and positive gains and the operational latency peak in order to schedule π_{new}. In this section, we introduce the algorithms.

Total Negative and Positive Gains Prediction. To predict Γ^- and Γ^+, the predicted processing latencies and inter-arrival times have to be combined. Each processing latency bin represents a number of events in π_{new} having a specific λ_p; each iat bin represents a number of events having a specific iat. In order to calculate the total negative and positive gain of all events, the number of events having a specific combination of λ_p and iat is predicted.

```
1:  ⟨long, long⟩ predictGains ( ) begin          ▷ returns Γ⁻ and Γ⁺
2:    predict #events for each latency bin B_l:  #(B_l)
3:    sort latency bins by mean latency (highest first)
4:    predict #events for each iat bin B_i:  #(B_i)
5:    sort iat bins by mean iat (lowest first)
6:    while true do
7:      #combination ← min{#(B_l), #(B_i)}
8:      gain ← #combination * (Θ̄ * λ_p^π(B_l) − iat(B_i))
9:      if gain > 0 then
10:       Γ⁻ ← Γ⁻ + gain
11:     else
12:       Γ⁺ ← Γ⁺ + gain
13:     end if
14:     #(B_l) ← #(B_l) − #combination
15:     #(B_i) ← #(B_i) − #combination
16:     if #(B_i) = 0 then
17:       i ← i + 1                              ▷ next iat bin
18:     end if
19:     if #(B_l) = 0 then
20:       l ← l + 1                              ▷ next latency bin
21:     end if
22:     if no more bins then
23:       return ⟨Γ⁻, Γ⁺⟩
24:     end if
25:   end while
26: end function
```

Figure 9: Predict Negative and Positive Gains.

```
1:  OperatorInstance ω_x                        ▷ current operator instance
2:  void schedule ( ) begin
3:    λ_o^max ← LatencyModel.newPrediction()
4:    if λ_o^max ≤ LB then
5:      assign σ to ω_x
6:    else
7:      x ← (x + 1) MOD #op_instances           ▷ Round-Robin
8:      assign σ to ω_x
9:    end if
10: end function
```

Figure 10: Scheduling algorithm.

To this end, events from the bin with highest λ_p are combined with the lowest iat, etc., and events with lowest λ_p are combined with the highest iat. The concrete algorithm is presented in the following (cf. algorithm in Figure 9). First, for each type, the total number of events, $\#(type)$, is divided into latency bins according to the weights of the bins: The number of events $\#(B_l)$ in a latency bin B_l is: $\#(B_l) = \#(type) * weight(B_l)$. Then, all latency bins of all event types are globally sorted by their mean processing latency (highest first). The iat bins are sorted by the mean iat (lowest iat first); the number of events $\#(B_i)$ in an iat bin B_i is computed based on the total number of events, n, and the weight of the bin, $\#(B_i) = n * weight(B_i)$. Then, the numbers of events in the processing latency bins and iat bins are combined such that the highest processing latencies are combined with the lowest iats. The algorithm iterates through the bins (lines 6 – 25): For the combination of current latency and iat bin, the gain of the events in this combination is calculated based on the processing latency and the iat of the bins. If the predicted gain is greater than 0, it is added to the total negative gains, else it is added to the total positive gains. Then, the next combination of bins is processed. When the iteration went through all bins, the resulting total negative and positive gains are returned.

Operational Latency Peak. The operational latency peak λ_o^{max} is predicted with the formulas introduced in Section 4.1, taking into account the predicted parameters as described in Section 4.2: $\lambda_o^{max} = \lambda_q^{max} + \lambda_p^{max}$, with $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$. In doing so, λ_p^{max} is predicted as the in-partition processing latency λ_p^{π} of the most expensive event type, in the most expensive latency bin, denoted as $max(\lambda_p^{\pi})$, combined with the average overlap: $\lambda_p^{max} = \overline{\Theta} * max(\lambda_p^{\pi})$.

Scheduling. When scheduling a new partition, the controller checks whether batching it to the same operator instance where the last partition was assigned to would lead to a violation of LB. The scheduling algorithm is listed in Figure 10. The latency model is queried for a prediction of the operational latency peak λ_o^{max} (line 3). The predicted λ_o^{max} is compared to LB and a batch scheduling decision is made accordingly: If $\lambda_o^{max} \leq LB$, the partition is assigned to the same instance as the last partition (lines 4–5); else, it is scheduled to the next operator instance according to the Round-Robin algorithm (lines 6–8).

Symbol	Parameter Description
iat	average inter-arrival time of events
b	batch size, i.e., number of subsequent partitions scheduled to same op. instance
ps	partition scope, i.e., temporal scope of a partition
Γ^-, Γ^+	total negative and positive gains
α	compensation factor
λ_o, λ_q, λ_p	operational latency, queuing latency and processing latency of an event in an operator instance; $\lambda_o = \lambda_q + \lambda_p$
λ_q^{max}	queuing latency peak: $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$
λ_q^{init}	initial queuing latency before processing the first event of a partition
LB	latency bound, i.e., the peak operational latency that shall not be exceeded
RR	Round-Robin scheduling, a non-batching scheduling algorithm that circularly assigns one partition to each operator instance
$\delta_{iat}, \delta_{\lambda_p}$	negative bias of measured iat or λ_p in the monitoring window, in std. deviations: e.g., $iat - \delta_{iat} * \sigma$
$mtime$	size of the workload monitoring window

Figure 11: Symbols used.

5. EVALUATION

In our evaluations, we analyze the proposed methods in two steps. In a first step, we perform a distinct evaluation of the proposed latency model. We show the accuracy and precision of the latency model in predicting the negative gains, positive gains and latency peaks in different situations under synthetic workloads. In the second step, we measure the performance of the overall event processing system under different realistic conditions—such as inter-arrival times, partition scopes, and latency bounds.

Experimental Setup and Notation. To evaluate the batch scheduling controller, we have integrated it into an existing data parallelization framework [26]. All experiments were performed on a computing cluster consisting of 16 homogeneous hosts with each 8 CPU cores (Intel(r) Xeon(R) CPU E5620 @ 2.40 GHz) and 24 GB memory, connected by 10-GB Ethernet links. The components of the parallelization framework were distributed among the available hosts.

Symbols used in the evaluations are listed in Figure 11.

5.1 Latency Model

In the following, we evaluate the accuracy and precision of the proposed latency model. We present the evaluation in two parts: First, we evaluate the predictions of the total negative and positive gains. In particular, we investigate how the proposed concepts of model refinements, i.e., using bins and standard deviations, influence the quality of predictions. Based on that, we then analyze the prediction of the queuing latency peak, which depends on the prediction of negative and positive gains as well as on the compensation factor α.

Interpretation of the figures in this section. We measured both the predicted values as well as the values that actually occurred in the operator instances. In all experiment results, on the y-axis, we depict the predicted values normalized to the measured values. For example, a value of 1.0 means that the prediction exactly met the actually occurred value, a value smaller than 1.0 means that the prediction was too low (i.e., underestimation), and a value higher than 1.0 means that the prediction was too high (i.e., overestimation). All figures depict the 10th, 25th, 50th, 75th, and 90th quantiles in a "candlesticks" representation.

5.1.1 Negative and Positive Gains

In analyzing the prediction of Γ^- and Γ^+, we run evaluations on synthetic workloads. Using synthetic workloads allows us to perform measurements in controlled situations where all of the parameters are well-known and completely under our control. This is not the case in real-world workloads, as we use them in the analysis of the overall event processing system in Section 5.2. For the face recognition operator, we created a synthetic stream of face events (i.e. images containing a person's face). Each 2 seconds, a burst of 4 face events with an inter-arrival time of 10 ms was created, which resembles a moderate workload (4 persons in front of a camera that captures a picture each 2 seconds). The query events were generated with a fixed rate of 1 query per second, so that each second, one new partition was started. For the traffic monitoring operator, we created a workload trace with an average inter-arrival time of events of 100 ms following an exponential distribution, which resembles a high workload (5 cars per second pass each road checkpoint).

Figure 12a shows evaluations of the face recognition operator at $b = 1$ using a different number of iat bins. If only 1 bin is used—i.e., the total mean iat is used in the latency model (cf. first strategy in Section 4.2)—the predictions of Γ^- and Γ^+ are poor. With a growing number of iat bins, the latency model becomes more accurate (cf. third strategy in Section 4.2): As can be seen, with 2, 4 or 8 bins, the predictions of both Γ^- and Γ^+ are very accurate and precise. 2 bins are sufficient, as the workload is also divided into two phases: face events arrive in bursts, and in between the bursts, no face events arrive. In contrast to the effect of iat bins, using a negative bias of δ_{iat} standard deviations (cf. second strategy in Section 4.2) does not make the predictions more accurate and precise, but more *pessimistic* (cf. 12b): The higher δ_{iat} is, the higher is the predicted Γ^-, but the lower is the predicted Γ^+. Further, we evaluate the impact of using bins and pessimistic bias for processing latency. As shown in Figures 12c and 12d, neither of the two strategies had positive impact on the accuracy of the latency model. This is because processing latency of single events in single partitions does not fluctuate very much in the face recognition operator.

We evaluated the latency model as well with the traffic monitoring operator. Same as in the face recognition operator, employing iat bins quickly improves the prediction accuracy (cf. Figure 13a). Further, using a negative bias of δ_{iat} standard deviations makes the prediction more pessimistic

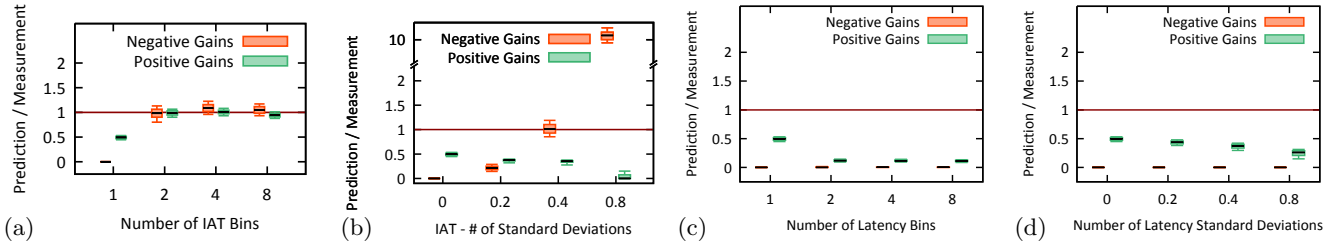


Figure 12: Face recognition operator at $ps = 10s$: prediction of negative and positive gains. $b = 1$.

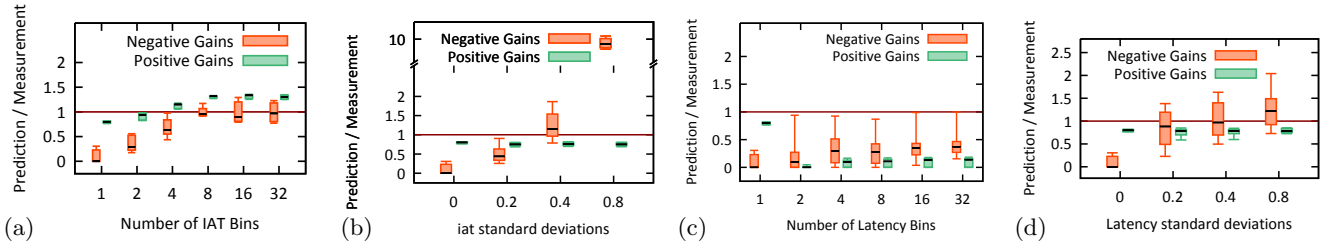


Figure 13: Traffic monitoring operator at $ps = 900s$: prediction of negative and positive gains. $b = 1000$.

(cf. 13b). Concerning processing latency, employing latency bins improves the accuracy of the latency model slightly (cf. Figure 13c). However, even though the processing latency in the traffic monitoring operator is position-dependent, the occurrence of negative and positive gains is still dominated by the *iat*; hence, the usage of latency bins alone does not lead to satisfactory results. In scenarios that show rather static *iat* and more heavily fluctuating latency, e.g., due to a position-dependency with super-linear impact, latency bins would have a stronger effect on the model quality. Employing a negative bias of δ_{λ_p} standard deviations on processing latency makes the latency model more pessimistic (cf. Figure 13d).

We also tested both scenarios with a higher batch size. In the face recognition operator at $b = 4$, employing *iat* bins leads to the same improvements of the model accuracy (cf. Figure 14a). In the traffic monitoring operator at $b = 2000$, even at only one *iat* bin, the accuracy is already high and adding more bins does not improve the model. This is because at a high batch size, the overlap of partitions is high, and hence, the processing latency of events—especially those of type *L2* (cf. Section 3.1)—is high as well. Basically, that means that most of the events of type *L2* will generate a negative gain, no matter if the specific *iat* of an event is high or low. The fluctuations of *iat* do not dominate the total negative and positive gains any longer; therefore, more *iat* bins do not improve the model.

We conclude, that in the tested scenarios, the accuracy of the latency model can be improved significantly by using *iat* bins when the occurrence of negative and positive gains is dominated by the *iat*—which is often the case. The strategies that employ negative bias on *iat* and processing latency can be used to make the model more pessimistic, which can be helpful in order to account for rapidly changing workloads.

5.1.2 *Queuing Latency Peak*

Recall that the queuing latency peak is predicted based on the total negative and positive gains and the compensation factor α: $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$. We show

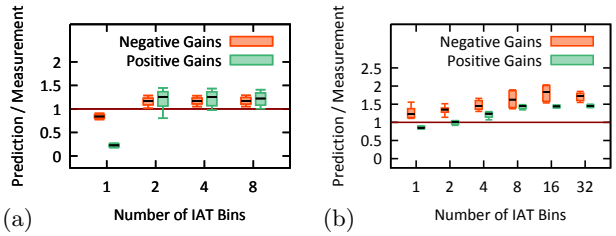


Figure 14: Higher batch sizes. (a) Face recognition, $b = 4$. (b) Traffic monitoring, $b = 2000$.

on the face recognition operator and the traffic monitoring operator that our proposed T-COUNT heuristic provides a suitable, slightly pessimistic estimation of α, such that no under-estimation of queuing latency peak occurs. Additionally, we evaluate the prediction of the initial queuing latency λ_q^{init}. Following our observations from Section 5.1.1, we employ the latency model with 2 *iat* bins for the face recognition operator, so that the predictions of Γ^- and Γ^+ are accurate.

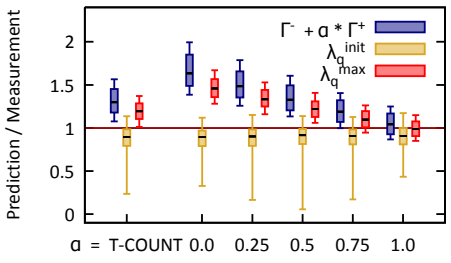


Figure 15: Predictions of queuing latency peak. Face recognition operator, $b = 4$, $ps = 10s$.

For the face recognition operator, we see in Figure 15 that the T-COUNT heuristics leads to a good overall estimation of λ_q^{max}. In predicting λ_q^{init}, there are some fluctuations, because events that are in the network and not yet arrived in the queue of an operator instance are not considered in the feedback to the splitter. However, the impact of this behav-

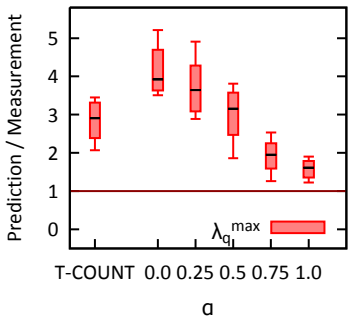


Figure 16: Predictions of queuing latency peak. Traffic monitoring operator, $b = 2000$, $ps = 900s$.

ior on the prediction of λ_q^{max} is small, as λ_q^{max} is dominated by the negative and positive gains. In the traffic monitoring operator, T-COUNT also guarantees that λ_q^{max} is not underestimated (cf. Figure 16). It is interesting that the measured λ_q^{init} was at 0 all the time. That means, when scheduling the 2000th partition in the batch, the incoming queue of an operator instance was still empty; hence, we omit the (trivial) prediction of λ_q^{init} in the figure. Queuing only happens when the events with high positions in the partitions of the batch are processed. This effect supports our argumentation from Section 3.2 that pure feedback-based scheduling is not applicable to the batch scheduling problem.

Besides the T-COUNT heuristic, we also systematically evaluated the impact of fixed values of α on the prediction of λ_q^{max} (cf. Figures 15 and 16). Using different fixed values leads to different degrees of over- or underestimations of λ_q^{max}. Off-line profiling can be used in order to develop optimally pessimistic or optimistic models to set α, when the characteristics of the workload are well-known before system deployment.

5.2 Overall Event Processing System

Batch scheduling imposes a temporary high load by assigning multiple subsequent partitions to the same operator instance, resulting in higher operational latency peaks, but reduced communication cost. To quantify the impact of batch scheduling on the performance of the overall event processing system, we compare batch scheduling to a scheduling algorithm that aims for good load balancing but disregards communication overhead: Round-Robin scheduling leads to minimal operational latency peaks in the operator instances, as the load and, hence, the operational latency, is evenly distributed among the operator instances. We evaluate the additional operational latency imposed by batching and the gains in terms of communication savings. Recall that our algorithm claims to be able to limit the additional operational latency imposed by the imbalanced scheduling to not exceed a latency bound LB. We also evaluate whether LB is kept even under challenging synthetic and *real-world workloads*.

Traffic Monitoring Scenario. In our dynamic traffic monitoring scenario, we modeled the inter-arrival time of vehicles as an exponential distribution with an average value following a sinusoidal curve between 2000 ms and 200 ms. According to the insights we gained from the evaluation of the latency model in Section 5.1, we set-up the controller to use 8 *iat* bins and $mtime = 60s$. To account for the position-dependency of the operator, we add a pessimistic bias of $\delta_{\lambda_p} = 2$ standard deviations on the monitored processing

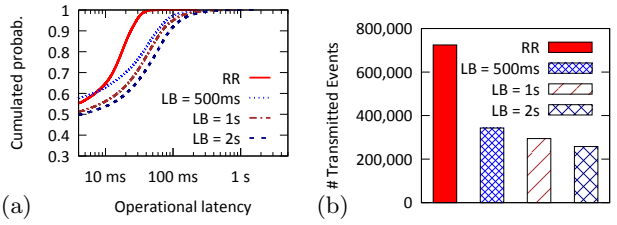


Figure 17: Traffic monitoring operator at $ps = 500s$. (a) Operational latency. (b) Communication cost.

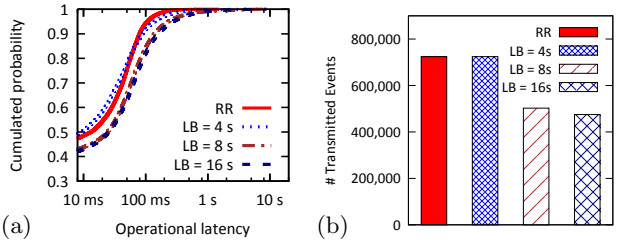


Figure 18: Traffic monitoring operator at $ps = 900s$. (a) Operational latency. (b) Communication cost.

latency. Further, we account for the rapidly changing *iat* by adding a pessimistic bias of $\delta_{iat} = 0.75$ standard deviations on the monitored *iat*. In all experiments, the parallelization degree, i.e., number of operator instances, was fixed at 8. Each experiment was running for 5 hours.

At a partition scope of 500 seconds, Round-Robin scheduling resulted in a maximal operational latency of 200 ms (cf. Figure 17a) and 724,464 events have been transmitted between the splitter and the operator instances (cf. Figure 17b). We ran the same experiment using our batch scheduling controller allowing for, 2.5, 5 and 10 times higher operational latency peaks than yielded in Round-Robin: 500 ms, 1 s and 2 s. As shown in Figure 17a, LB was kept. ***The communication overhead was reduced by 53 %, 59 % and 64 %, respectively*** (cf. Figure 17b).

At a partition scope of 900 seconds, Round-Robin scheduling resulted in a maximal operational latency of 1.6 s (cf. Figure 18a) and 724,272 events have been transmitted between the splitter and the operator instances (cf. Figure 18b). We ran the same experiment using our batch scheduling controller allowing for, 2.5, 5 and 10 times higher operational latency peaks than yielded in Round-Robin: 4 s, 8 s and 16 s. As shown in Figure 17a, LB was kept. The communication overhead was not reduced at $LB = 4s$. Due to the higher partition sizes, the achieved batching sizes were too small to avoid the problem of redundant event transmission to all operator instances. When LB is relaxed to 8 s and 16 s, ***communication overhead was reduced by 31 % and 34 %, respectively*** (cf. Figure 18b).

In summary, the batch scheduling controller kept the requested latency bounds throughout all experiments. We can significantly save communication cost by using batch scheduling if we allow for higher operational latency peaks than the optimal ones achieved with Round-Robin scheduling. Hence, we can trade operational latency for communication cost, and limit that trade-off by setting LB.

Face Recognition Scenario. With the dynamic face recognition scenario, we evaluate the behavior of batch scheduling in a highly bursty *real-world workload*. A *real video stream* from a camera installed on campus—capturing 1

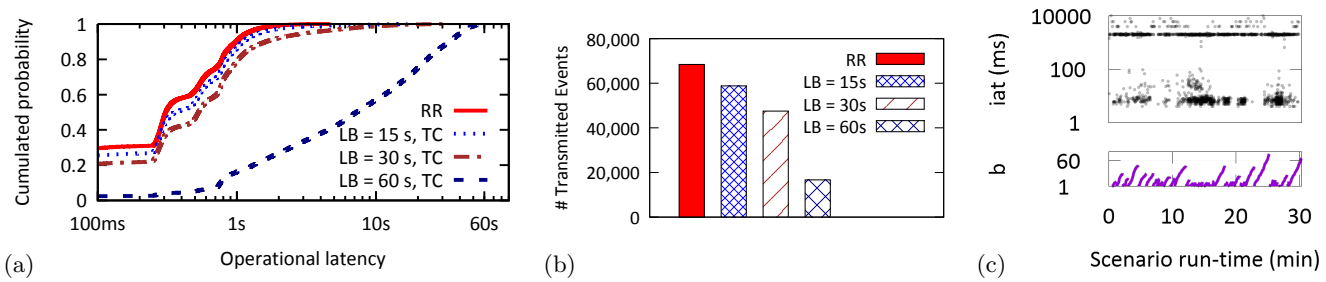


Figure 19: Face recognition operator. (a) Operational latency. (b) Communication cost. (c) Adaptation.

frame each 2 seconds—is processed by a face detection operator and the detected faces are streamed to the face recognition operator. Simulating users of a face recognition application, the arrival of new queries is modeled as an exponential distribution with an average inter-arrival time of 2 seconds. The face recognition operator detects whether the queried person is in the face event stream, using a partition scope of $ps = 10s$. Each experiment ran for 150 minutes. According to the insights we gained from the evaluation of the latency model in Section 5.1, we set-up the controller to use 2 *iat* bins. Further, we set $mtime = 10s$ and $\delta_{iat} = 1.0$ standard deviations to account for the rapidly changing *iat*.

For Round-Robin scheduling, we measured an operational latency peak of 6 seconds (cf. Figure 19a) and 68,412 events have been transmitted between the splitter and the operator instances (cf. Figure 19b). We ran the same experiment using our batch scheduling controller allowing for 2.5, 5 and 10 times higher operational latency peaks than yielded in Round-Robin: 15 s, 30 s and 60 s. The latency bounds are kept in all tested settings (cf. Figure 19a). ***The communication overhead was reduced by 14 %, 31 % and 76 %, respectively*** (cf. Figure 19b).

In Figure 19c, we visualize the workload; in contrast to many synthetic workloads, it is very bursty and unpredictable. We see how the batch size is continuously adapted to the highly bursty workload, without any prior training. We conclude, that even in very bursty workloads, we can trade operational latency against communication cost using the proposed batch scheduling controller.

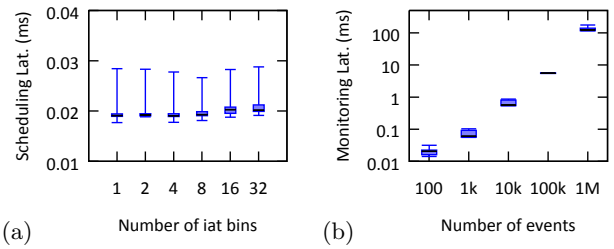


Figure 20: Latency of (a) scheduling (traffic operator), (b) updating statistics.

Scalability. We evaluate the scalability of our approach in two aspects. First, the *scheduling latency*, i.e., the time between the detection of the start of a new partition and the decision which operator instance the partition should be assigned to (cf. Algorithm in Figure 10). It includes predicting the negative and positive gains (cf. Algorithm in Figure 9), whose complexity is determined by the granularity of the latency model, i.e., the number of bins used in the model. We measured a very low scheduling latency of in average 0.02 ms for up to 32 bins used (cf. Figure 20a), which is the maximal number of bins needed in any of the scenarios that we have tested (cf. Section 5.1.1).

Second, we evaluate the time needed to update the latency model with new statistics from the monitoring window, i.e., the *monitoring latency*. This includes recomputing the weights, average values and standard deviations of the bins. Using 32 bins, we measured that the monitoring latency grows linearly with the number of events in the monitoring window (cf. Figure 20b). At 1,000,000 events in a monitoring window, updating the statistics took between 100 and 200 ms, allowing the statistics still to be updated 5 times a second even when using such large monitoring windows.

6. RELATED WORK

Over the last two decades, Complex Event Processing (CEP) has evolved as the paradigm of choice to detect and integrate events in situation-aware applications [15, 1, 12, 30]. Recently, the research focus has been shifted from centralized CEP systems [9, 14] towards Distributed CEP (DCEP) middleware [20, 4, 28], pushing the operators close to the sources. However, individual operators can be a bottleneck and operator parallelization is needed [19, 11, 26]. Besides data parallelization, intra-operator parallelization—also known as pipelining [5, 11]—has been proposed, which depends on the functional parallelism of an operator and, hence, is limited in its scalability.

In existing DCEP data parallelization frameworks, scheduling has not been in the focus of attention. In the related field of stream processing systems, there have been addressed different problems of assigning batches of data to instances of stream processing operators. Das et al. [17] propose a reactive controller in order to batch a minimal number of events to an operator such that the throughput is sufficiently high to process the current workload. In their processing model, operators can aggregate larger chunks of data more efficiently, so that the throughput of operators grows with the batch size. A similar problem had been studied before by Carney et al. [10]. Unlike in this paper, in all of these systems, partitions scheduled to operator instances are not overlapping and all events of a batch are known when the batch is scheduled. Balkesen and Tatbul [6] recognize the trade-off of communication overhead to latency in operator instances when scheduling overlapping partitions. They introduce an analytical cost model which assumes fixed processing latency of an event in a partition and fixed count-based or time-based partition size and slide. Further, their cost model does not consider inter-arrival times. Hence, the model is not suitable for solving the batch scheduling problem in data-parallel DCEP operators. In the work of Li

et al. [25], the cost of evaluation sliding-window aggregation queries is reduced by sub-aggregating the events in non-overlapping subsets and sharing computation steps. This is not applicable to pattern detection in data-parallel DCEP operators, where each operator instance needs to receive the complete set of events in order to detect a queried pattern.

Existing work about scheduling in non-parallel DCEP middleware and similar systems does not take into account the network utilization, but tries to optimize the usage of other resources like CPU [21] and memory [3]. Classical single- and multiprocessor scheduling algorithms, like Round-Robin, FIFO, SPN, Deadline-Monotonic Scheduling, etc., have different optimization goals and do not regard batching of overlapping data sets.

Other latency models for DCEP operators have been proposed. The Mace metrics from Chandramouli et al. [13] for latency estimation in a DCEP middleware proposes an analytical model. However, it assumes the usage of their proposed scheduling algorithm—which is not a batch scheduling algorithm. In the latency model of Zeitler and Risch, a fixed processing latency of each event is assumed [32]; our latency model differentiates between different event types and takes into account the overlap of partitions.

Batching has also drawn attention in many other fields, like graph processing [29, 31] and column data-stores [24, 8]. In such problems, it is often preferable to process or store data in batches instead of handling each single tuple separately. However, typically, optimal batch sizes are predefined, e.g., by cache sizes, so that fixed batch sizes are employed. In publish/subscribe middleware, the minimization of the communication cost for aggregation in event brokers has been studied [27]. However, this is constrained to aggregation in sliding windows; a DCEP middleware allows for more expressive operations.

7. CONCLUSION

In this paper, we have pointed out the trade-off between communication overhead and latency when batch scheduling subsequent partitions in data-parallel DCEP operators. We have evaluated key factors determining the operational latency in operator instances. As the batch scheduling decisions are made on open partitions, a long feedback delay between the decisions and their impact on feedback parameters is induced, making reactive scheduling approaches infeasible. Instead, we have proposed a model-based controller. We have evaluated the latency model and shown that the controller batches an optimal amount of partitions even at bursty workloads. This way, the problem of high communication overhead can be mitigated.

8. REFERENCES

[1] A. Adi and O. Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[2] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. SIGMOD '05, pages 13–24. ACM, 2005.

[5] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul. RIP: Run-based intra-query parallelism for scalable complex event processing. DEBS '13, pages 3–14. ACM, 2013.

[6] C. Balkesen and N. Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. International Workshop on Data Management for Sensor Networks (DMSN), 2011.

[7] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. SIGCOMM '11, pages 242–253. ACM, 2011.

[8] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, volume 5, pages 225–237, 2005.

[9] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. VLDB '02, pages 215–226. VLDB Endowment, 2002.

[10] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker. Operator scheduling in a data stream manager. VLDB '03, pages 838–849. VLDB Endowment, 2003.

[11] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. SIGMOD '13, pages 725–736. ACM, 2013.

[12] S. Chakravarthy and D. Mishra. Snoop: An expressive event specification language for active databases. *Data Knowl. Eng.*, 14(1):1–26, 1994.

[13] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos. Accurate latency estimation in a distributed event processing system. ICDE '11, pages 255–266. IEEE, 2011.

[14] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. SIGMOD '03, pages 668–668. ACM, 2003.

[15] G. Cugola and A. Margara. Tesla: a formally defined event specification language. DEBS '10, pages 50–61. ACM, 2010.

[16] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[17] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. SOCC '14, pages 16:1–16:13. ACM, 2014.

[18] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. SIGCOMM '09, pages 51–62. ACM, 2009.

[19] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Comput. Surv.*, 46(4):46:1–46:34, Mar. 2014.

[20] N. Jain, L. Amini, H. Andrade, R. King, Y. Park, P. Selo, and C. Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. SIGMOD '06, pages 431–442. ACM, 2006.

[21] L. Kencl and J.-Y. Le Boudec. Adaptive load sharing for network processors. *IEEE/ACM Trans. Netw.*, 16(2):293–306, Apr. 2008.

[22] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz. Rollback-recovery without checkpoints in distributed event processing

systems. DEBS '13, pages 27–38. ACM, 2013.

[23] K. LaCurts, S. Deng, A. Goyal, and H. Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 Internet Measurement Conference*, IMC '13, pages 191–204, New York, NY, USA, 2013. ACM.

[24] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, Aug. 2012.

[25] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, Mar. 2005.

[26] R. Mayer, B. Koldehofe, and K. Rothermel. Predictable low-latency event detection with parallel complex event processing. *Internet of Things Journal, IEEE*, 2(4):274–286, Aug 2015.

[27] N. K. Pandey, K. Zhang, S. Weiss, H.-A. Jacobsen, and R. Vitenberg. Minimizing the Communication Cost of Aggregation in Publish/Subscribe Systems. In *35th IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2015.

[28] N. P. Schultz-Møller, M. Migliavacca, and P. Pietzuch. Distributed complex event processing with query rewriting. DEBS '09, pages 4:1–4:12. ACM, 2009.

[29] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *Proc. VLDB Endow.*, 7(3):193–204, Nov. 2013.

[30] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. SIGMOD '06, pages 407–418. ACM, 2006.

[31] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *Proc. VLDB Endow.*, 6(14):2014–2025, Sept. 2013.

[32] E. Zeitler and T. Risch. Massive scale-out of expensive continuous queries. *VLDB Endowment*, 4(11), 2011.