Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing*

Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel
{ruben.mayer,adnan.tariq,kurt.rothermel}@ipvs.uni-stuttgart.de
Institute for Parallel and Distributed Systems
University of Stuttgart, Stuttgart, Germany

ABSTRACT

Distributed Complex Event Processing has emerged as a well-established paradigm to detect situations of interest from basic sensor streams, building an operator graph between sensors and applications. In order to detect event patterns that correspond to situations of interest, each operator correlates events on its incoming streams according to a sliding window mechanism. To increase the throughput of an operator, different windows can be assigned to different operator instances—i.e., identical operator copies—which process them in parallel. This implies that events that are part of multiple overlapping windows are replicated to different operator instances. The communication overhead of replicating the events can be reduced by assigning overlapping windows to the same operator instance. However, this imposes a higher processing load on the single operator instance, possibly overloading it. In this paper, we address the trade-off between processing load and communication overhead when assigning overlapping windows to a single operator instance. Controlling the trade-off is challenging and cannot be solved with traditional reactive methods. To this end, we propose a model-based batch scheduling controller building on prediction. Evaluations show that our approach is able to significantly save bandwidth, while keeping a user-defined latency bound in the operator instances.

CCS CONCEPTS

•Computer systems organization →Distributed architectures;

KEYWORDS

Complex Event Processing, Data Parallelization, Communication Overhead

ACM Reference format:
Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing. In *Proceedings of ACM International Conference on Distributed and Event-Based Systems, Barcelona, Spain, June 19 - 23, 2017 (DEBS '17),* 12 pages.
DOI: http://dx.doi.org/10.1145/3093742.3093914

*Supported by Deutsche Forschungsgemeinschaft (DFG), project grant "PRECEPT".

DEBS '17, Barcelona, Spain

978-1-4503-5065-5/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3093742.3093914

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, 25] 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.

In a DCEP system, high workload on the operators can lead to overload and long buffering delays when they process incoming streams only sequentially. To increase the operator throughput, data parallelization [4, 23] has been proposed as a powerful parallelization method. In a data parallelization framework, incoming event streams of an operator are split into windows 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 window comprises all events needed in order to detect a pattern. This means that different windows can overlap, i.e., events are part of multiple windows [4, 23].

When splitting incoming event streams, the data parallelization framework assigns a window to an operator instance when the start of the window is detected. In doing so, assigning overlapping windows to different operator instances results in increasing communication overhead, as events that are part of multiple different windows are replicated to multiple operator instances. In the worst case, an event may be transmitted to all operator instances, leading to a high network load. In cloud data centers, this may not only impair the performance of the hosted DCEP systems, but also the performance of other applications hosted on the same infrastructure. Network-intensive applications have been identified as a major cause of bottlenecks in cloud data centers [6, 14, 19]. Therefore, reducing the bandwidth consumption of parallel DCEP systems can be of great worth to all hosted applications.

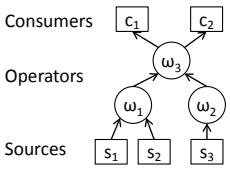


Figure 1: DCEP operator graph.

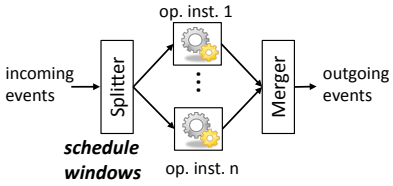


Figure 2: Data parallelization framework.

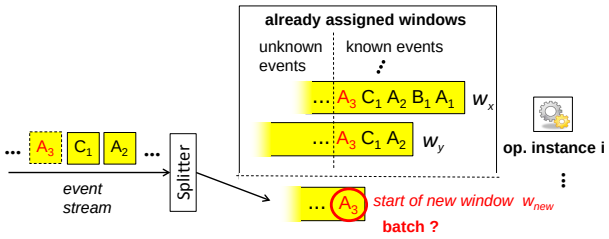


Figure 3: Splitting and scheduling.

To reduce the bandwidth consumption, we employ batch scheduling of subsequent overlapping windows, i.e., assigning them to the same operator instance. That way, events from the overlap only need to be transferred once. However, at the same time, the operator instance must process more windows 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 optimal amount of windows, which cannot be solved with state-of-the-art scheduling algorithms from stream processing [5, 7, 21].

- **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*.
- **Window overlap**: The overlap between windows of a batch influences the processing load induced by each event, as each event is processed in the context of each window it is part of. Moreover, the scheduling decision is made on open windows, i.e., the events and the overlap of a window 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 windows. (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 common model of a data-parallel DCEP system [4, 23].

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, an operator ω detects event patterns in finite, non-empty subsets of their incoming event streams—called *windows* and denoted by w, using a correlation function $f_\omega : w \rightarrow (e_1, e_2, ..., e_n)$. The patterns to be detected can be defined in event specification languages like Snoop [9] or Tesla [11], e.g., sequence patterns, logical (AND / OR / NOT) patterns, and others. Pattern definitions take into account event meta-data such as timestamps and event types, but can also rely on user-defined functions that analyze the events' payload, e.g., face recognition functions.

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 [4, 23]. A splitter divides the incoming event streams of the operator into windows. The windows are then scheduled (i.e., assigned) to an elastic set of operator instances which simultaneously process their assigned windows. Finally, a merger orders the events emitted by the operator instances into a deterministic sequence. Each window must comprise all events needed in order to detect a pattern instance.

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 [9]. The splitter opens a window whenever an event of type A occurs, and closes the window after one minute. The operator instances check whether in a window, events of type B and C occur in the right order. Taking a look at the splitting, we see that all events following A_1 within one minute are part of the same window w_x: If some of the events would be missing, they could not be checked for the Sequence(B;C) sub-pattern that follows A_1.

In contrast to horizontal splitting, vertical splitting techniques (e.g., "panes" [21] or "stream batches" [18]) have been proposed for stream aggregation operators. For instance, when the max or median value of a window of 1 minute shall be computed, that window could be split into 6 fragments of 10 seconds, the fragments' max or median be computed in parallel, and the global window's value be computed from the local results. This way, mere aggregation functions can be efficiently computed, as processing and aggregating the event subsets is an embarrassingly parallel task. In DCEP

[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.

pattern detection, the processing of any event may depend on the complete temporal history of preceding events. For instance, when detecting a sequence of three events A, B, and C, processing of an event depends on the other events that have been detected before. Often, additional constraints are formulated, e.g., that B and C must occur within one minute from A (as in the example above), or that A, B and C have a parameter x, such that $A.x < B.x < C.x$ (e.g., to detect chart patterns in stock markets [4]). This naturally leads to a window-based processing model, where windows capture the dependencies between different event sets that *potentially* build a pattern match; the windows themselves can hardly be split into smaller fragments, because the dependencies between the events may span the complete window. Hence, horizontal splitting is a natural choice to exploit the data parallelism in such pattern detections (cf. [11, 23]).

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 interconnected 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 definition, windows can have different sizes and a different number of events can occur between two start events of subsequent windows. We denote the period of time that a window spans, i.e., the time between the first event and the last event of a window, as the *window scope*, *ws*. Further, we denote the period of time between two start events of subsequent windows as the *window shift*, Δ. Upon detection of the start of a new window, this window is assigned to an operator instance according to a scheduling algorithm. In an operator instance, incoming events are processed sequentially. Within each window, an event has a different context. Therefore, when processing an event e, the operator instance sequentially processes e in the context of each window that e is part of.

Example: Recall the scenario in Figure 3. Two overlapping windows w_x and w_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 w_x than in w_y: In w_x, the sequence (B;C) is detected, while in w_y, the sequence is not detected. In checking the occurrences of the sequence pattern in different windows, operator instance i processes C_1 sequentially first in w_x and then in w_y.

From the event consumer's point of view, the situation detection latency is the period from the occurrence of a source 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: violations of the latency bound shall, if possible, be avoided. The situation detection latency spans the whole operator graph and is sub-divided into latency budgets for each single operator. In each operator, the splitter and the merger induce latency for splitting the streams into windows and merging the results. Because scheduling windows to operator instances significantly influences the latency induced in each operator instance, in this paper, we focus on batch scheduling suitable amounts of windows 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 windows 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 windows. Overall, the operational latency of an event is a combination of its queuing latency and processing latency, i.e., $\lambda_o(e) = \lambda_q(e) + \lambda_p(e)$.

Problem Formalization: To minimize the communication overhead, the batch scheduling controller tries to assign as many subsequent windows as possible to the same operator instance subject to the constraint that the operational latency of events in that instance must not exceed a latency bound *LB*. As soon as the start of a new window w_{new} is detected by the splitter, the batch scheduling controller decides whether assigning that window to the same operator instance as the previous window would cause operational latency of events to exceed *LB*. This is noted as the *batch scheduling problem* in data-parallel DCEP operators.

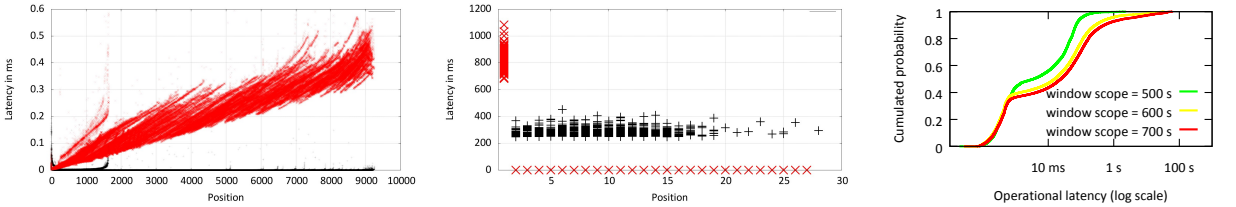
Example: The trade-off tackled in the batch scheduling problem is exemplified in Figure 3. An event A_3 arrives at the splitter and the splitter detects that A_3 starts a new window w_{new} which now has to be scheduled. Suppose a set of previous windows W_{old} =(..., w_x, w_y) has already been scheduled to a specific operator instance i. Events before A_3 in W_{old} have been transferred to operator instance i. However, further events arriving after A_3 can as well be part of some of the windows in W_{old}; hence, they are transferred to operator instance i, too. When scheduling w_{new} to operator instance i, communication overhead can be reduced, because events overlapping between w_{new} and W_{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 w_{new}, inducing higher processing latency. The splitter has to decide whether w_{new} can be assigned to operator instance i such that the operational latency does not increase beyond *LB*.

3 BATCH SCHEDULING

To analyze the batch scheduling 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. Then, in Section 3.2, we highlight the difficulties in 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* window. Based on that, we identify and analyze key factors that influence operational latency in a whole *batch* of windows. To this



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

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

(c) Traffic monitoring: Operational latency with reactive batch scheduling at $TH = 100ms$ under different window scopes.

Figure 4: Evaluations.

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* window window: Whenever a vehicle a passes $L1$, a window w is opened, and when the same vehicle passes $L2$, w is closed. Another vehicle b that appears in the $L1$ stream within w has passed $L1$ after a. When b appears again in w 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 [9] language as an aperiodic operator: Aperiodic(A; B; C) with A $\rightarrow \langle$plate=a, type=$L1\rangle$, B $\rightarrow$ Sequence($\langle$plate=b, type=$L1\rangle$;$\langle$plate=b, type=$L2\rangle$), C $\rightarrow \langle$plate=a; type=$L2\rangle$[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 $\rightarrow \langle$type=*query*, time=$t\rangle$, B $\rightarrow$ $\langle$type=*face*, "`face_match(A)`"$\rangle$, C $\rightarrow$ *time* $\geq t +$ *time frame*.

Processing Latency of Events in a Window. When processing a *single* window in an operator instance, each event imposes a specific processing latency. This is different from stream processing where the processing latency of an event in a window is considered fixed [5, 29]. We identified two key factors that influence the processing latency of an event in a window: its type and its position.

Event type. Event types are a fundamental concept in DCEP. Many query languages, such as Snoop [9], Amit [1], SASE [26] and Tesla [11], 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 window (cf. Figure 4b). In both operators, we see different processing latencies depending on the event types.

Position of Event. When processing events of a window, internal state is gathered in an operator [4, 8], 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($\langle$plate=b, type=$L1\rangle$;$\langle$plate=b, type=$L2\rangle$) or close the window. Therefore, e_{L2} is compared to all *L1* events that have been seen in the window before (*equi-join* operator). 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 a query event; the `face_match` function imposes the same processing latency in each event position (cf. Figure 4b).

Operational Latency in a Batch of Windows. In a *batch* of windows, different windows may overlap. When the batch scheduling controller assigns a window to an operator instance that overlaps with other windows, the processing latency of all events in the overlap is influenced, as events are processed sequentially in the scope of their windows. Recall that a window has to comprise all events needed in order to detect a queried pattern. Therefore, the overlap of different windows 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 [12, 21].

In the following, we identify key factors influencing the overlap of windows 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, using 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

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

scenario parameters				measurements			
#	batch size	avg. iat (s)	*ws* (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)	*ws* (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.

experiment, more than 370,000 operational latency measurements have been taken.

Batch size. The batch size, i.e., number of windows assigned to an operator instance in a batch, influences the overlap of the windows, and hence, the operational latency of events. 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, #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 windows, e.g., in time-based windows. The number of events in windows 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, #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).

Window scope (ws). The window scope *ws*—i.e., the time between the start and end event of a window—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 [9, 11]. For instance, in the traffic monitoring operator, the start and end of a window depend on the speed of the vehicles, as a window 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 window opened from this vehicle is larger. Therefore, the size and overlap of windows can change even when the batch size and *iat* stay the same. This is different from stream processing, where only windows of fixed size and fixed slide— time- or count-based—are analyzed [5]. In the traffic monitoring operator, we increased *ws* 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, #5, #6 and #7). In the face recognition operator, however, increasing *ws* led to a proportional increase in operational latency peaks.

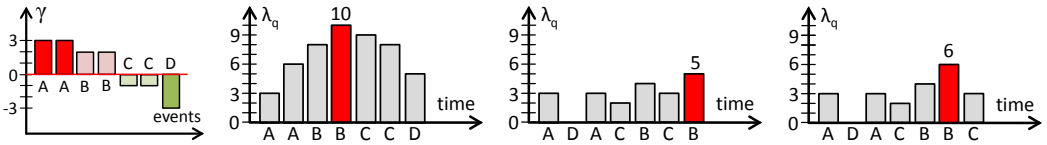
From the observations on key factors that influence operational latency when processing a batch of windows, we see that building a direct mapping from *batch size*, *inter-arrival time* and *window scope* to operational latency peaks in operator instances is hard. The relation between key factors and operational latency peaks that occur in operator instances is complex, and different in different operators. A model trained before run-time (off-line), 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 [12, 21]. The basic idea of a reactive controller is that it schedules windows according to *feedback parameters* (like operational latency or queue length) from the operator instances that indicate how many windows 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 window to which instance this window is scheduled. Then, it directs all events that arrive in the scope of that window to the corresponding instance. It is infeasible for the controller to wait until all events of the window are present and then schedule the window; it would take too much time in view of per-event latency bounds. After assigning a window to an operator instance at the occurrence of its start event, many other events of that window arrive until the window is finally closed. Thus, over the *whole time span* of the window, feedback parameters in the operator instance are influenced by the scheduling decision, i.e., a long time after



(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.

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 [12], 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 window to an operator instance and the occurrence of the peak value of the feedback parameters in that operator instance. 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, window scope, event types, etc. continuously change. Moreover, 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 devise a *latency-reactive* controller that reacts on the *current* operational latency in operator instances. Windows are batched to the same operator instance until at the instance, the current operational latency reaches a threshold *TH*; subsequent windows are scheduled to the next operator instance. This, however, poses the question how to set *TH*. A simple experiment shows that a static *TH* is not good enough to keep the latency bound *LB*. 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 $TH = 100ms$, reactive batch scheduling more or less was able to keep *LB* when *ws* was not higher than 500 s (cf. Figure 4c). However, at a *ws* of 600 s and 700 s, $TH = 100ms$ 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 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: The feedback delay is high. Again, using the current queue length as feedback parameter requires a suitable threshold, which in turn has to be adapted to changing key factor values.

In the face recognition operator, window 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, 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

The batch scheduling controller must predict whether the operational latency peak in an operator instance will be higher than *LB* when batching a new window w_{new}. To this, we introduce a latency model. We aim to find the right balance between the complexity, 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 window contains 7 events: 2 events of type A impose λ_p = 8 TU, 2 events of type B impose λ_p = 7 TU, 2 events of type C impose λ_p = 4 TU, and 1 event of type D imposes λ_p = 2 TU. Then, the gains of the single events are between +3 and -3 TU (+3 for type A, +2 for B, -1 for C, -3 for D).

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

[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 window in a batch.

termed as the *total positive gain*[4]: $\Gamma^+ = \sum \gamma(e) : e \in w_{new} \wedge \lambda_p(e) < iat$. 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 w_{new} is very difficult to predict. It would essentially correspond to predicting each single event in w_{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 w_{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 w_{new} might already face a queuing latency λ_q^{init} at its arrival. This can be due to previous windows that had been scheduled to the same operator instance. Hence, the final formula to calculate the 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 w_{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 w_{new}; i.e., it does not regard individual events, but it regards events in w_{new} as *sets* of events imposing negative or positive gains. Hence, it builds on the prediction of the *set* of events in w_{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.

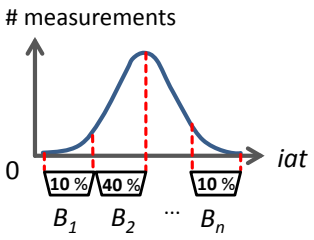


Figure 7: *iat* bins.

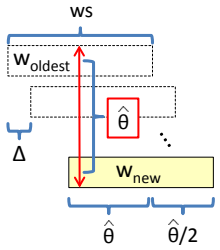


Figure 8: Overlap.

Inter-arrival time. The splitter continuously monitors the past *iat* values in a window of *mtime* time units. Our *iat* model tackles two challenges: heavy fluctuations of the *iat* around an average value (variance) and rapid changes of the average *iat* (changing trend).

Tackling the first challenge, the splitter arranges the monitored inter-arrival times in a discrete model (cf. Figure 7). 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 $\overline{iat(B_i)}'$ is computed: $iat(B_i) = \overline{iat(B_i)}'$. Each bin is assigned with a $weight(B_i)$, i.e. ratio of number of entries in the bin to total number of measurements in all bins. The number of bins manages the accuracy of the model; the minimum number of bins is 1.

Tackling the second challenge, we introduce a negative bias on the monitored mean value $\overline{iat(B_i)}'$ in each bin. This way, the model accounts for changes in the average *iat* between the monitored value $\overline{iat(B_i)}'$ and the value that will occur in w_{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, $iat(B_i) = \overline{iat(B_i)}' - \delta_{iat} * \sigma$.

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

For modeling $\lambda_p^w(type)$, we propose the same methods as for modeling *iat*, using a combination of negative bias and bins. Same as in *iat* bins, in each latency bin B_l, we predict $\lambda_p^w(B_l) = \overline{\lambda_p^w(B_i)}' + \delta_{\lambda_p} * \sigma$, i.e., the measured mean in-window processing latency in the latency bin plus a factor δ_{λ_p} of standard deviations. The advantage of monitoring the current (distribution of) $\lambda_p^w(type)$ 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 windows change, e.g., due to

[4] If $\lambda_p(e) = iat$, neither negative nor positive gains occur.
[5] For the sake of readability, we did not mention in the text that $\lambda_q^{max} = \lambda_q^{init}$, if $\Gamma^- + \alpha * \Gamma^+ < 0$.

changing workload or changing window scopes, this is reflected in the monitored current (distribution of) $\lambda_p^{\mathrm{w}}(type)$ values. We do not need to explicitly model the positions of individual events.

The overlap Θ for all events of w_{new} is modeled as the average overlap of events of w_{new} in the current batch, denoted by $\overline{\Theta}$. Predicting $\overline{\Theta}$ is performed according to the following model (cf. Figure 8). When w_{new} is scheduled in a batch of already opened windows, a number of events in w_{new} has the current overlap $\hat{\Theta}$, until the oldest open window w_{oldest} in the batch closes. From closing w_{oldest} until closing w_{new}, the overlap decreases step-wise in regular intervals each time a window between w_{oldest} and w_{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 windows in the batch have the same window scope ws, and between the start of two windows there is the same shift Δ; ws and Δ are measured in the splitter at regular intervals to keep them up to date at each scheduling decision.

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

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

Number of Events. For predicting the set of events in w_{new}, there are three significant factors in the model: (1) The window scope ws, (2) the iat, and (3) the ratio of different event types, denoted as $ratio(type)$, that models which percentage of events in w_{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. To predict the total number of events in w_{new}, we again use a negative bias of δ_{iat} standard deviations $\sigma(iat)$, so that $iat = \overline{iat}' - \delta_{iat} * \sigma(iat)$. Then, the total number of events n is predicted as $n = \frac{ws}{iat}$, 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 windows in regular intervals to the splitter. The splitter calculates λ_q^{init} of an operator instance as the sum of the processing latencies of all reported events in its queue: $\lambda_q^{init} = \sum_{types} \#events * \overline{\Theta} * \lambda_p^{\mathrm{w}}(type)$.

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-window processing latency λ_p^{w}: the group of events with higher λ_p^{w} is denoted by T^- and the group of events with lower λ_p^{w} is denoted by T^+. The distinction between the groups is made based on the average $\lambda_p^{\mathrm{w}}(type)$ of the event types; there is one half of event types that has higher $\lambda_p^{\mathrm{w}}(type)$ 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.

```
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^w(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_l) = 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.

4.3 Scheduling Algorithm

Having a prediction of the set of events in w_{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 w_{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 w_{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. 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 *iat*s. 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-window processing latency λ_p^{w} of the most expensive event type in the most expensive latency bin, denoted as $max(\lambda_p^{\mathrm{w}})$, at the average overlap: $\lambda_p^{max} = \overline{\Theta} * max(\lambda_p^{\mathrm{w}})$.

Scheduling. When scheduling a new window, the controller checks whether batching it to the same operator instance where the last window 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 window is assigned to the same instance as the last window (lines 4–5); else, it is scheduled to the next operator instance according to the Round-Robin algorithm (lines 6–8).

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 and latency bounds—comparing the model-based batch scheduling controller to Round-Robin and to a reactive batch scheduling algorithm. The cost of prediction is also evaluated.

Experimental Setup and Notation. To evaluate the batch scheduling controller, we have integrated it into an existing data parallelization framework [23]. 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.

Symbol	Parameter Description
iat	average inter-arrival time of events
b	batch size, i.e., number of subsequent windows scheduled to same op. instance
ws	window scope, i.e., temporal scope of a window
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 window
LB	latency bound, i.e., the peak operational latency that shall not be exceeded
RR	Round-Robin scheduling, circularly assigns one window to each operator instance
δ_{iat}, δ_{λ_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
TH	scheduling threshold of reactive baseline controller, cf. Section 3.2

Figure 11: Symbols used.

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. 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, i.e., Figure 12 and Figure 13, 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 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 window 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 5 cars per second passing each road checkpoint.

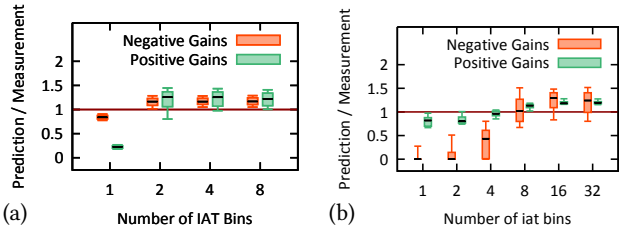


Figure 12: Prediction of negative and positive gains. (a) Face recognition operator, $b = 4$, $ws = 10s$, (b) Traffic monitoring operator, $b = 1000$, $ws = 500s$.

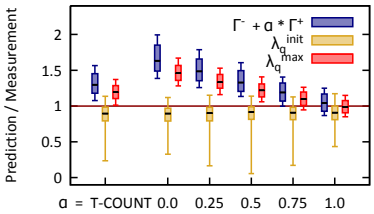


Figure 13: Predictions of queuing latency peak. Face recognition operator, $b = 4$, $ws = 10s$.

Figure 12a shows evaluations of the face recognition operator using a different number of *iat* bins. If only 1 bin is used, the predictions of Γ^- and Γ^+ are poor. With a growing number of *iat* bins, the latency model becomes more accurate: With 2, 4 or 8 bins, the predictions of both Γ^- and Γ^+ are very accurate and precise. In the traffic monitoring operator, employing more *iat* bins, as shown in the results in Figure 12b, quickly improves the prediction quality as well.

We have also evaluated the effects of a negative bias on *iat*, as well as bins and negative bias on processing latency; a detailed discussion with all the results can be found in [24]. Summarizing those results, using a negative bias of δ_{iat} standard deviations in the *iat* bins makes the model more pessimistic. Further, the negative and positive gains in the tested scenarios are dominated by *iat* such that employing bins and negative bias only on processing latency is insufficient to tune the model.

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 on the example of the face recognition 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, so that the predictions of Γ^- and Γ^+ are accurate.

We see in Figure 13 that the T-COUNT heuristics leads to a good overall estimation of λ_q^{max}. In predicting λ_q^{init}, fluctuations are caused by events in the network that have not yet arrived in the queue of an operator instance and are not considered in the feedback to the splitter. However, the impact of this behavior on the prediction of λ_q^{max} is small, as λ_q^{max} is dominated by the negative and positive gains.

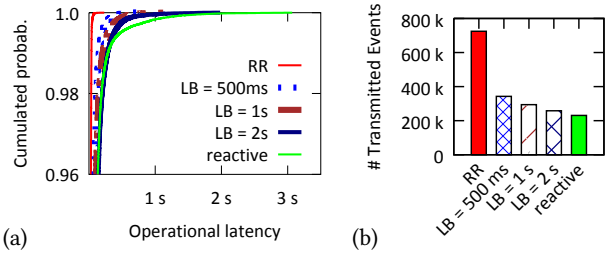


Figure 14: Traffic monitoring operator. (a) Operational latency. (b) Communication cost.

Besides the T-COUNT heuristic, we also systematically evaluated the impact of fixed values of α on the prediction of λ_q^{max}. As can be seen in Figure 13, 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

We compare our model-based batch scheduling controller to two baseline scheduling algorithms: Round-Robin scheduling and latency-reactive scheduling. Round-Robin aims for good load balancing but disregards communication overhead; it is the standard scheduling algorithm used in window-based data parallelization systems such as [23]. Latency-reactive scheduling, as described in 3.2, batches windows to an operator instance until its operational latency exceeds a threshold *TH*. It is used as a latency-aware baseline algorithm to compete against our model-based controller.

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. Following the evaluation of the latency model in Section 5.1, we set-up the controller to use 8 *iat* bins and a tumbling monitoring window with $mtime = 60s$. To account for the position-dependency of the operator and the rapdily changing workload, we add a pessimistic bias of $\delta_{\lambda_p} = 2$ standard deviations on the monitored processing latency and $\delta_{iat} = 0.75$ 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 window scope of 500 seconds, Round-Robin scheduling resulted in a maximal operational latency of 200 ms (cf. Figure 14a) and 724,464 events have been transmitted between the splitter and the operator instances (cf. Figure 14b). 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 14a, *LB* was kept. ***The communication overhead was reduced by 53 %, 59 % and 64 %, respectively*** (cf. Figure 14b). We compared this performance to the latency-reactive scheduler described in Section 3.2; the reactive scheduler batches windows to an operator instance until it reports a current operational latency of more than $TH = 100ms$. The operational latency and communication overhead was very similar to model-based scheduling at $LB = 2s$; however, the tail of the latency distribution is much longer, leading to 50 % higher

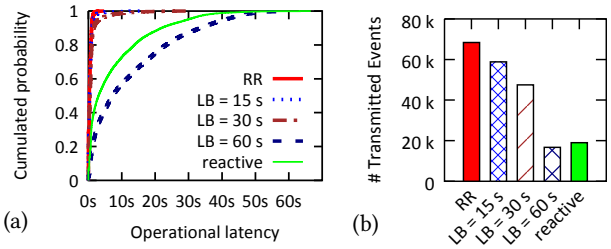


Figure 15: Face recognition operator. (a) Operational latency. (b) Communication cost.

operational latency peaks. This indicates that the reactive scheduler erratically batches too many windows, leading to a less predictable behavior of the operator instances than when the model-based controller is used.

Face Recognition Scenario. With the dynamic face recognition scenario, we evaluate the system behavior at a highly bursty *real-world workload.* A *real video stream* from a camera installed on campus—capturing 1 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 window scope of $ws = 10s$. 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 $mtime = 10s$ (tumbling window) and $\delta_{iat} = 1.0$ standard deviations to account for the changing *iat*.

For Round-Robin scheduling, we measured an operational latency peak of 6 seconds (cf. Figure 15a) and 68,412 events have been transmitted between the splitter and the operator instances (cf. Figure 15b). 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 15a). ***The communication overhead was reduced by 14 %, 31 % and 76 %, respectively*** (cf. Figure 15b). We compared this performance to the latency-reactive scheduler described in Section 3.2 with $TH = 6s$. The operational latency peaks were 15 % higher than with the model-based controller at $LB = 60s$, while the communication overhead was 14 % higher as well. With a higher threshold *TH*, the reactive scheduler would induce even higher latency peaks, while with a lower *TH*, it would induce an even higher communication overhead; hence, the model-based controller is more effective, no matter how the reactive scheduler's threshold is set up.

In summary, model-based batch scheduling is effective in trading communication overhead against operational latency. In comparison, reactive scheduling is less predictable and effective than model-based scheduling; it might still be useful in cases where a simple best-effort batching approach is sufficient, but should not be used when latency bounds must be enforced.

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 window and the scheduling decision (cf. Algorithm in Figure 10). It includes predicting the negative

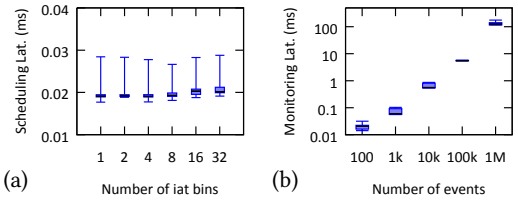


Figure 16: Latency of (a) scheduling and (b) updating statistics.

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 16a), which is the maximal number of bins needed in any of the scenarios that we have tested (cf. Section 5.1.1). For comparison, the median scheduling latency in reactive scheduling and in Round-Robin scheduling was both 0.004 ms. Admittedly, there is a small overhead for the model-based batch scheduling controller involved compared to the simple strategies. This is not significant in most scenarios; if scheduling would be a throughput bottleneck in the splitter, the frequency of predicting negative and positive gains could be adapted (i.e., not predicting fresh gains at each single scheduling decision), trading model accuracy against throughput.

Second, we evaluate the time needed to update the latency model with new statistics from the monitoring window, i.e., the *monitoring latency*. This comprises recomputing the weights, average values and standard deviations of the bins. Using 32 bins, we measured a linear growth with the number of events in the monitoring window (cf. Figure 16b). At 1,000,000 events in a monitoring window, updating the statistics took between 100 and 200 ms, which is a reasonable time to adapt the model to changes in the workload.

6 RELATED WORK

Complex Event Processing (CEP) has evolved as the paradigm of choice to detect and integrate events in situation-aware applications [1, 9, 11, 26]. In doing so, distributed CEP (DCEP) systems [16, 25] distribute the detection logic over a network of operators. However, individual operators can be a bottleneck and operator parallelization is needed [8, 23]. Besides data parallelization, intra-operator parallelization—also known as pipelining or state-based parallelization [4]—has been proposed, which is limited by the functional parallelism of an operator. Besides window-based splitting as presumed in this paper, key-based splitting [8, 15] has been proposed, that is splitting by a key that is encoded in the events. However, this is limited to the number of different key values, e.g., different stock symbols in an algorithmic trading scenario. Moreover, not all DCEP patterns exhibit key-based data parallelism, whereas window-based data parallelism is inherent to most of them, as DCEP operators in their very nature work on windows (cf. [11, 23]).

In related work, there have been addressed different problems of assigning batches of individual events to instances of stream processing operators. Das et al. [12] 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 sets of events more efficiently, so that the throughput of operators

grows with the batch size. A similar problem had been studied before by Carney et al. [7]. Micro-batching, as used, e.g., in Spark Streaming [28], provides efficient failure recovery and batch-like programming paradigms by handling streaming events as a series of fixed-sized batches. Unlike in this paper, in all of these approaches, batches are composed of individual events and not of overlapping windows. Balkesen and Tatbul [5] recognize the trade-off of communication overhead to latency in operator instances when scheduling overlapping windows. Their analytical cost model assumes fixed processing latency of an event in a window and fixed count-based or time-based window size and slide. Further, it does not consider inter-arrival times. Hence, it is not suitable for solving the batch scheduling problem in data-parallel DCEP operators.

Elasticity in data-parallel stream processing, i.e., adapting the number of operator instances to changing workloads, is a complementary problem. Existing solutions that apply latency models often base on the assumption of fair load balancing [13, 22, 23]; batch scheduling defeats this assumption, deliberately inducing a controlled load imbalance. How to use the proposed latency model of the batch scheduling controller for elasticity control is an interesting research question for future work.

Other latency models for DCEP operators have been proposed. The Mace metrics from Chandramouli et al. [10] 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 [29]; our latency model differentiates between different event types and takes into account the overlap of windows.

Batching is also applied in other fields, like graph processing [27] and column data-stores [20], where 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.

Scheduling algorithms in non-parallel DCEP optimize the usage of resources like CPU and memory [3, 17] without taking into account batching of overlapping data sets.

7 CONCLUSION

In this paper, we have tackled the problem to batch as many subsequent overlapping windows as possible to the same operator instance in data-parallel DCEP operators subject to the constraint that the operational latency in the operator instance must not exceed a given latency bound. As the batch scheduling decisions are made on open windows, 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. Evaluations show that the controller batches an optimal amount of windows even at bursty workloads. This way, the bandwidth consumption of data-parallel DCEP operators can be significantly reduced.

REFERENCES

[1] Asaf Adi and Opher Etzion. 2004. Amit - the Situation Manager. *The VLDB Journal* 13, 2 (May 2004), 177–203.

[2] Arvind Arasu, Shivnath Babu, and Jennifer Widom. 2006. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal* 15, 2 (June 2006), 121–142.

[3] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. 2004. Operator Scheduling in Data Stream Systems. *The VLDB Journal* 13, 4 (Dec. 2004), 333–353.

[4] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. RIP: Run-based intra-query parallelism for scalable complex event processing *(DEBS '13)*. ACM, 3–14.

[5] Cagri Balkesen and Nesime Tatbul. 2011. Scalable data partitioning techniques for parallel sliding window processing over data streams *(International Workshop on Data Management for Sensor Networks (DMSN))*.

[6] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. 2011. Towards Predictable Datacenter Networks *(SIGCOMM '11)*. 242–253.

[7] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. 2003. Operator Scheduling in a Data Stream Manager *(VLDB '03)*. VLDB Endowment, 838–849.

[8] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management *(SIGMOD '13)*. ACM, 725–736.

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

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

[11] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: a formally defined event specification language *(DEBS '10)*. ACM, 50–61.

[12] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. 2014. Adaptive Stream Processing Using Dynamic Batch Sizing *(SOCC '14)*. ACM, Article 16, 13 pages.

[13] Tiziano De Matteis and Gabriele Mencagli. 2016. Keep Calm and React with Foresight: Strategies for Low-latency and Energy-efficient Elastic Data Stream Processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '16)*. ACM, New York, NY, USA, Article 13, 12 pages.

[14] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A Scalable and Flexible Data Center Network *(SIGCOMM '09)*. 51–62.

[15] Martin Hirzel. 2012. Partition and Compose: Parallel Complex Event Processing *(DEBS '12)*. 191–200.

[16] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. 2006. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core *(SIGMOD '06)*. ACM, 431–442.

[17] Lukas Kencl and Jean-Yves Le Boudec. 2008. Adaptive Load Sharing for Network Processors. *IEEE/ACM Trans. Netw.* 16, 2 (April 2008), 293–306.

[18] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. 2016. SABER: Window-Based Hybrid Stream Processing for Heterogeneous Architectures *(SIGMOD '16)*. 555–569.

[19] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. 2013. Choreo: Network-aware Task Placement for Cloud Applications. In *Proceedings of the 2013 Internet Measurement Conference (IMC '13)*. 191–204.

[20] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1790–1801.

[21] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams. *SIGMOD Rec.* 34, 1 (March 2005), 39–44.

[22] Björn Lohrmann, Peter Janacik, and Odej Kao. 2015. Elastic Stream Processing with Latency Guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*. 399–410.

[23] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. 2015. Predictable Low-Latency Event Detection with Parallel Complex Event Processing. *Internet of Things Journal, IEEE* 2, 4 (Aug 2015), 274–286.

[24] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. *Real-time Batch Scheduling in Data-Parallel Complex Event Processing.* Technical Report 2016/04. University of Stuttgart. 14 pages.

[25] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed Complex Event Processing with Query Rewriting *(DEBS '09)*. Article 4, 12 pages.

[26] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams *(SIGMOD '06)*. 407–418.

[27] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. 2013. Fast Iterative Graph Computation with Block Updates. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 2014–2025.

[28] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. 2012. Discretized Streams: An Efficient and Fault-tolerant Model for Stream Processing on Large Clusters *(HotCloud'12)*. USENIX Association.

[29] Erik Zeitler and Tore Risch. 2011. Massive scale-out of expensive continuous queries. *VLDB Endowment* 4, 11 (2011), 1181–1188.