Predictable Low-Latency Event Detection with Parallel Complex Event Processing

Ruben Mayer*, Boris Koldehofe† and Kurt Rothermel*
*Institute for Parallel and Distributed Systems
University of Stuttgart, Germany
Email: firstname.lastname@ipvs.uni-stuttgart.de
†Multimedia Communications Lab
Technische Universität Darmstadt, Germany
Email: Boris.Koldehofe@kom.tu-darmstadt.de

***Abstract*—The tremendous number of sensors and smart objects being deployed in the Internet of Things pose the potential for IT systems to detect and react to live-situations. For using this hidden potential, Complex Event Processing (CEP) systems offer means to efficiently detect event patterns (complex events) in the sensor streams and therefore help in realizing a "distributed intelligence" in the Internet of Things. With the increasing number of data sources and the increasing volume at which data is produced, parallelization of event detection is crucial to limit the time events need to be buffered before they actually can be processed. In this article, we propose a pattern-sensitive partitioning model for data streams that is capable of achieving a high degree of parallelism in detecting event patterns which formerly could only be consistently detected in a sequential manner or at a low parallelization degree. Moreover, we propose methods to dynamically adapt the parallelization degree to limit the buffering imposed on event detection in the presence of dynamic changes to the workload. Extensive evaluations of the system behavior show that the proposed partitioning model allows for a high degree of parallelism and that the proposed adaptation methods are able to meet a buffering limit for event detection under high and dynamic workloads.**

***Keywords*—Complex Event Processing, Stream Processing, Data Parallelization, Self-Adaptation, Quality of Service**

I. Introduction

Nowadays, billions of sensors and *smart objects*, i.e., objects with embedded electronics that enable identification, sensing and actuation capabilities, are deployed throughout the globe, collecting data about the physical world, like smart meters, GPS sensors and RFID tags. They grow in numbers [1] and have the power to enable new applications in the areas of smart homes, smart cities, environmental monitoring, healthcare, smart business and security, paving the way towards the *Internet of Things* (IoT) [2], [3]. In doing so, many applications encompass a high and fluctuating number of entities or events to be monitored. For instance, in smart grid applications like demand response [4], [5], hundreds of thousands of events per second are emitted from millions of consumers, with relevant load changes varying by up to a factor of 7 over time [6]. Similar fluctuations can be observed in traffic monitoring: Official traffic statistics from the California Department of Transportation[1] show that in one single hour ("rush hour") up to 25 % of the total daily traffic volume can occur on streets.

[1]http://traffic-counts.dot.ca.gov/

In order to support automated, timely reactions of smart objects to situations of interest occurring in the physical world, an *intelligence* is needed that enables their consistent detection by integrating and continuously analyzing low-level sensor streams. Consistency in this regard means that neither false-positives nor false-negatives should be detected. *Complex Event Processing (CEP)* [7]–[11] is a key paradigm that helps in realizing such an intelligence. It allows for specifying a graph of multiple dependent operators that step-wise transform low-level sensor data to complex events that correspond to the situations of interest, e.g., critical power grid situations or traffic jams. The CEP systems further provide means to execute the operators in a distributed manner to ensure an efficient utilization of the available resources.

However, traditional CEP systems are not yet capable of supporting low latency event detection for large-scale IoT applications. Delays in event detection can have several causes, such as delays imposed by communication, processing, and buffering of events. While for communication and processing latencies, bounds can be found that can be met with high probability, unlimited buffering delays can occur when an operator is overloaded and therefore needs to buffer an unlimited amount of events. In this regard, current CEP systems are not providing parallelization methods that can yield sufficient throughput and lack methods to adapt their resources to workload fluctuations. Thus, they cannot guarantee keeping a buffer limit and this way lack supporting low latency event detection. This shortcoming can have drastic consequences and dramatically reduces the benefits an IoT application can draw from a CEP system. Therefore, it is of critical importance to develop CEP systems that can guarantee a buffer limit even under high and fluctuating workloads.

In this article, we extend our findings in [12] in order to make CEP operators capable of keeping a predictable buffer limit. Our contributions are threefold: (i) We propose a novel *pattern-sensitive* stream partitioning model. The partitioning model allows to consistently parallelize a wide class of CEP operators and ensures a high degree of parallelism. (ii) We propose methods to model the workload and dynamically adapt the parallelization degree utilizing *Queuing Theory (QT)*, so that a buffering limit of each operator can be met predictably. Our evaluation shows that the proposed stream partitioning methods can achieve a high throughput of up to 380,000 events per second even on commodity hardware. (iii) Moreover, we

Copyright (c) 2015 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.
This article has been published in the IEEE Internet of Things Journal. http://dx.doi.org/10.1109/JIOT.2015.2397316

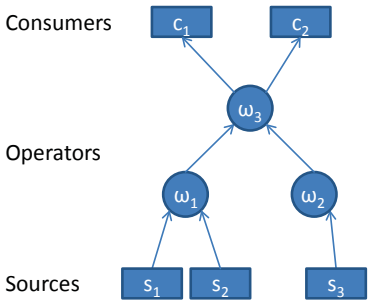


Fig. 1. Operator graph with sources, operators, and consumers, connected by event streams.

show in the context of a traffic monitoring scenario that the adaptation methods enforce even under heavily fluctuating workloads a stable parallelization degree and this way ensure that the buffering limit is met and only little over-provisioning in terms of resources is required.

The article is structured as follows. Section II gives an overview of the development of CEP systems over the last decade, their impact on IoT applications and the shortcomings of current approaches with regard to low latency event detection. Section III describes the problem that we tackle with the system and formalizes the guarantees that it provides. The novel stream partitioning model is described in Section IV. In Section V, the problem of adapting the operator parallelization degree is tackled. In Section VI, we provide an extensive evaluation of the system, showing the performance and efficiency in the context of exemplary realistic scenarios in the field of traffic monitoring. Finally, we conclude the article and give an outlook on future work in Section VII.

II. Background

In this section, we first introduce the notion of a CEP system. We explain the benefits and challenges of CEP in the context of IoT applications. We conclude that only parallel CEP systems can be capable of supporting low latency event detection in large-scale scenarios. Finally, we analyze the existing parallel CEP approaches and show their shortcomings in providing low latency event detection.

A. CEP Systems

Complex event processing (CEP) [7]–[9] has evolved as the paradigm of choice to detect and integrate events in situation-aware applications. In CEP, domain experts specify the events to be detected in a *query language* like Snoop [13] or TESLA [9]. The CEP system then deploys an operator graph [14]–[16], incorporating event sources, operators and event consumers—i.e., applications interested in the queried events—that are interconnected by event streams. An *event* e contains a tuple of attribute-value pairs (the *payload* of the event), a unique *sequence number* within its stream and a time stamp. Events of different streams inherently possess a well-defined global order by their time stamps, sequence numbers and source. The set of events in all incoming or outgoing streams of an operator ω is denoted I_ω or O_ω respectively.

In Figure 1, an operator graph is depicted. Basing on the *query*, each operator defines how to detect event patterns

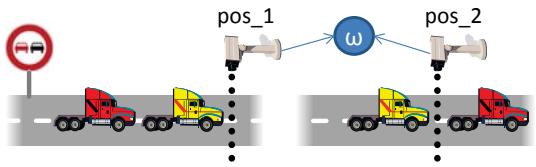


Fig. 2. Smart city: Surveillance of a no-passing zone utilizing two sensors.

on its incoming event streams and specifies the events to be produced whenever an event pattern was detected. To this end, an operator performs a sequence of independent *correlation steps* on the events of the incoming streams [15]. In each step, a finite, non-empty set of events denoted *selection* σ is correlated and a finite tuple of events is produced and emitted to the successor. This can be represented by a mapping of tuples of incoming events $\mathbb{E}_{in}$ to tuples of outgoing events $\mathbb{E}_{out}$, which is denoted the *correlation function* $f_\omega : \mathbb{E}_{in} \mapsto \mathbb{E}_{out}$. When correlating events of a selection, the operator executes *computations* on the events in σ. Such computations can be as simple as applying a filter rule or as complex as running a face recognition algorithm on a set of video frames that are contained in the event payload. This way, the operator graph detects complex situations step-wise from the basic sensor streams and emits the corresponding complex events to the consumers.

B. Benefits and Challenges of CEP in IoT Scenarios

Applications in the area of the Internet of Things are becoming increasingly complex and therefore pose special requirements on CEP. To optimize processes, for example, in logistics, the smart grid [4], [5], or traffic management [17]–[20], it is necessary to support such applications to react to the occurrence of critical situations with low latency. Such situations can, for instance, be accidents, load variations in power consumption, or changing weather conditions. The monitoring of situations of interest is enhanced by a steadily growing infrastructure of globally deployed sensors. By continuously capturing and analyzing sensor streams, CEP systems can help to detect the relevant situations.

When applications react on events, they trigger decisions like updating a traffic route or changing the configuration in power consumption. Therefore, the utility of such decisions is strongly dependent on i) the time at which events are delivered by the CEP system, and ii) the view of the application on the set of delivered events at decision time. This view should be consistent, i.e., at time of decision all relevant events are present (no false-negatives) and no false events (no false-positives) are delivered [15].

For instance, consider an application from the area of smart cities, a Traffic Monitoring System (TMS) as depicted in Figure 2. On highways, it is often desirable to establish an overtaking ban, especially in danger spots like road construction sites. Given two camera-based sensors deployed at the beginning and end of the no-passing zone (pos_1 and pos_2), an operator ω can detect when a vehicle overtakes another one. This application apparently requires both the consistent and the low latency detection of complex events that signal when a vehicle has overtaken another one. A false-positive

detection would cause an unjustified ticket, while a false-negative detection would leave the transgressor unpunished. The low latency detection of an overtaking maneuver allows, among other things, for direct feedback to drivers who have violated the traffic rules. Note that the operator is part of a larger operator graph, which is not depicted in Figure 2, for instance, operators that identify the vehicles from the camera streams. Also those operators require consistent and low latency event detection, as inconsistencies and delays are propagated to operator ω.

In a managed environment like a data center it is in many cases easy to find acceptable bounds for communication and processing latencies that can be met with high probability. For instance, in the traffic monitoring operator from Figure 2, we have measured in 99 % of the cases a maximal delay of 1.25 seconds for processing 1 event (cf. Section VI-B). However, understanding communication and processing delays is not enough when the operator suffers from overload, i.e., the arrival rate of events exceeds the achievable processing rate. In this case, two situations can occur: (i) The operator applies *load shedding*, simply discarding events that it is not capable of processing in time [21], [22]. This apparently leads to inconsistencies, which are not tolerable in many IoT applications. (ii) A high amount of events need to be buffered before being processed, which can cause an unacceptable latency in event detection. For instance, in the former traffic monitoring operator, several thousand buffered events induce a buffering delay in the magnitude of minutes or even hours. Thus, in order to support low latency event detection, it is crucial to keep a buffering limit in the operators.

This is highly challenging since many applications like traffic monitoring applications and demand response optimizations in smart grids encompass very high and fluctuating event rates. Efficient and consistent parallel execution of dozens of processing entities is necessary to speed up CEP systems such that they are capable of meeting the buffering limits. High and fluctuating event rates pose also a significant cost in terms of resource utilization. In the above example, the processing capabilities of up to 8 processing entities are necessary to keep up with the maximum arrival rate at operator ω (cf. Section VI). When being deployed for the worst-case workload, the resources are underutilized for a long time of the day when the traffic intensity is lower, causing unnecessary cost for resource provisioning. The cost is amplified by the fact that applications typically react to dozens, if not hundreds of different patterns that can occur in the smart city, each of them requiring the deployment of a set of distinct operators. Changing the parallelization degree of operators dynamically according to the current workload—given by the arrival rate of events—is expected to yield significant cost gains by avoiding underutilization of the system resources. Therefore, CEP operators need to support, in addition to parallelization, dynamic reconfigurations and adaptive provisioning of resources with sufficient capabilities such that the operator can meet a buffering limit for a given workload at minimal cost.

C. Parallel CEP

In the following, we analyze state-of-the-art parallel CEP systems and investigate whether they are capable of supporting consistent low latency event detection for a wide range of operators, for example aperiodic, period and sequence operators as defined in the operator specification language Snoop [13]. In doing so, we analyze the ability to consistently reach high processing rates through operator parallelization, and the ability to dynamically change the parallelization degree when facing fluctuating workloads, so that a buffer limit is kept at low cost.

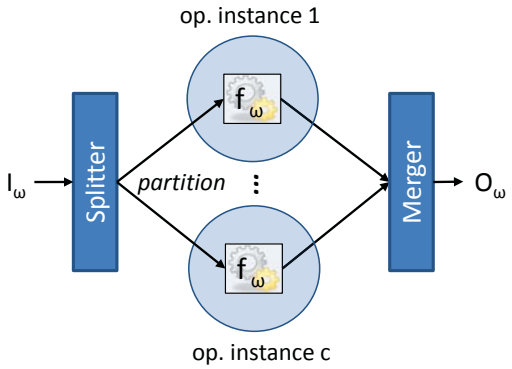


Fig. 3. Data parallelization framework.

1) Intra-operator parallelization: In intra-operator parallelization, also known as pipelining, internal processing steps that can be run in parallel are identified by deriving operator states and transitions from the query [23], [24]. The operator logic is split accordingly, and the identified processing steps are executed in parallel on the incoming event streams. This approach offers only a limited achievable parallelization degree depending on the number of variables in the query. For instance, operator ω from the scenario in Figure 2 only offers four variables.

To show why this is the case, we look a bit deeper into the operator. Each time a vehicle x passes one of the sensors pos_1 or pos_2, a source event x_1 or x_2 is created. To detect violations on the overtaking ban, operator ω utilizes a *window*: When a vehicle a passes pos_1 (a_1), a window is opened, and when it passes pos_2 (a_2), the window is closed. Another vehicle b that appears in the pos_1 stream (b_1) within that window has passed pos_1 **after** a. When b appears again in the same window in the pos_2 stream (b_2), it has passed pos_2 **before** a. If this is the case, b has overtaken a and thus violated the traffic rules. To detect this situation, the following query is applied, e.g., in Snoop [13] notation: Sequence(a_1;b_1;b_2;a_2), which results in a maximal parallelization degree of four. This degree is insufficient for coping with high event loads, as is proven by our evaluations in Section VI.

Thus, intra-operator parallelization is not sufficient in that case, and a parallelization approach is needed that is capable of yielding a higher degree of parallelism.

2) Data parallelization: A more promising approach is *data parallelization* [24]–[33] by applying stream partitioning [34]. In this approach, instead of splitting the operator logic as in intra-operator parallelization, the incoming event streams are split into partitions that can be processed by a number of identical instances of the operator. Figure 3 depicts the architecture of such a framework, consisting of a splitter, a number of operator instances and a merger. The splitter

assigns the events from I_ω according to a *partitioning model* to different operator instances. The execution of an operator instance is controlled by a runtime environment (RE). This comprises for the main part the management of partitions and communication with the splitter. The RE receives information about the assigned partitions and the corresponding events, and manages the operator execution so that the assigned partitions are processed. The merger ensures that an ordering between all produced events is established, if such an ordering is required at subsequent operators or event consumers. The ordered events are consecutively assigned sequence numbers.

Two different partitioning models have been proposed in data parallelization: key-based and batch-based. *Key-based* partitioning, that is partitioning by a key that is encoded in an event, is applied in almost all current data parallelization approaches [25]–[33]. The parallelization degree is limited to the number of different key values, if a common key is available at all, and the membership of a distinct event to a certain pattern must not depend on the appearance of other events, which is the case in the operator from our TMS scenario. Thus, key-based partitioning only offers limited expressiveness. *Batch-based* stream fission, as proposed in the run-based parallelization approach in [24], proposes to split the streams into batches that are large enough to fit any match to a queried pattern. Besides the problem that this can cause communication overhead when patterns fluctuate in their size in terms of comprised events, the approach is insufficient to support consistent partitioning for operators that detect patterns of an unknown size. This is the case in the operator from our TMS scenario and also in many other operators, e.g., in aperiodic, periodic and sequence operators [13].

Concluding from the observations we have made so far, there is currently no consistent operator parallelization method that allows for a high degree of parallelism for a large class of relevant CEP operators.

3) Parallelization degree adaptation: When facing fluctuating workloads, the parallelization degree must be continuously adapted in order to keep a buffering limit at low resource cost. However, since fluctuations of the workload can have a delayed effect on the imposed processing efforts in event detection, state-of-the-art approaches [35], [36] that only react to changes in the utilization of resources are not able to ensure a predictable buffering limit. As later also confirmed by our evaluation results, those reactive approaches impose to exceed the acceptable buffering limit at times by a factor of more than 1000.

The reactive scale-out approach in [35] proposes scaling out based on the current CPU load of operator instances. However, the approach does not offer any guarantees on meeting a specific buffering level, as our evaluation results also have shown. In [36], a more sophisticated parallelization degree adaptation algorithm is proposed that bases on the parameters congestion and throughput. However, it does not guarantee buffer limits on bursty workloads and does not account for the problem of large selections that have a delayed effect on the operator, as it is the case in our traffic monitoring operator. [37] tries to forecast the exact event arrival rate and assumes a fixed per-tuple processing cost when determining the optimal parallelization degree, which does not always hold, as we show with the operator profiles in our traffic monitoring scenario (cf. Section VI). Performance modeling approaches, e.g., Queuing Petri Nets [38] or latency estimation metrics like Mace [39], require a deeper knowledge of and interference with the operators and do not provide methods to adapt the parallelization degree in order to yield a limited buffering level at fluctuating workloads.

III. Problem Description

Although CEP is an important paradigm to enable situation-aware IoT applications, state-of-the-art methods neither consistently reach a high parallelization degree nor can adapt to fluctuating workloads in a way that a buffer limit is guaranteed. As we have motivated in Section II, there is a need for corresponding methods, so that IoT can fully benefit from CEP in many areas.

In this section, we formalize the problem of keeping a predictable buffer limit. To this end, we first specify the system assumptions and go into details about how a buffer in a CEP operator works. Based on that setup, we specify the goals our framework needs to reach in order to allow for low latency event detection.

A. System and Queue Model

a) System Model: The data parallelization framework is deployed on an infrastructure that consists of a number of computing nodes that are considered failure-free and provide a homogeneous computing capability, i.e., the same CPU and memory capabilities[2]. The number of hosts that can be used by the data parallelization framework is flexible and a sufficient number of hosts is available. This way, new hosts can be allocated for the deployment of operator instances as well as deallocated when they are not used any more. The allocation of a new host and deployment of an operator instance takes TH *(Time Horizon)* time units from the allocation request until the instance is available. The hosts are connected by communication links which guarantee eventual in-order delivery of data.

b) Queue Model: Events arriving on the incoming streams of the splitter are stored in-order in a queue. The instances process the events of the assigned partitions, and acknowledge them as soon as they are processed. When an event has been acknowledged by all operator instances which it has been assigned to, the event is consumed, i.e., discarded from the splitter queue. This way, the queue grows or shrinks depending on the ratio of acknowledged events to newly arriving events.

B. Requirements

Under this setup, for an operator ω, the following guarantees must be provided: Given a correct prediction of the *probabilistic distribution* of inter-arrival times of events TH time units in the future, the parallelization degree is constantly adapted so that a user-defined buffering level limit BL_ω of

[2]To work with heterogeneous nodes, the methods we developed in this article can be extended to take into account individual computational capabilities of nodes by utilizing operator profiles for all available node types (cf. Section V-B). However, homogeneous capabilities are a common case in virtualized computing environments, so that the assumption is practical to focus on the main challenges that are tackled in this article.

buffered events in the queue of the splitter will be kept with a user-defined probability $P_{required}$. For example, if the buffer limit is 100 with $P_{required}$ = 95 %, then the filling level of the queue at the splitter will be below 100 events with a probability of 95 % when checked at an arbitrary point in time.

This involves solving two sub problems: Consistent stream partitioning and the adaptation of the parallelization degree to fluctuating workloads. The splitter should work with a simple model that only requires minimal knowledge about the operator logic that is needed in order to partition the incoming streams. Executing the operator in the parallelization framework should be possible without interference into the operator logic. To increase the processing rate of an operator, stream partitioning in the splitter should happen independently of processing the partitions in the operator instances. Furthermore, the operator instances should process the assigned partitions while sharing only minimal state amongst each other and needing only minimal synchronization effort. The method for adapting the parallelization degree has to take into account the time needed to deploy a new operator instance on a processing entity.

IV. Stream Partitioning

To find partitions in the event streams that yield consistent processing results in the operator instances without requiring any adaptations of the operator logic, based on the event processing model introduced in Section II-A we develop a *partitioning model* that is capable to partition the streams in such a way that the operator instances produce consistent streams according to the event processing model.

A. Partitioning Model

Recap that an operator is executed according to its *correlation function* $f_\omega : \mathbb{E}_{in} \mapsto \mathbb{E}_{out}$, mapping selections of events from the incoming streams to detected events that are emitted on the outgoing streams. As the correlation function is a mapping, between two correlation steps no computational state is maintained. That means that two selections can be processed on different hosts independent of each other.

The idea of the proposed partitioning model is to split the incoming event stream *by selections* which contain the patterns to be detected, so that we refer to the model as *pattern-sensitive stream partitioning*. That means that each partition must comprise one or more complete selections, i.e., all events that are part of the selection(s).

To ensure that a selection is completely comprised in a partition, all events between the first and the last event of the selection must be part of that partition. Thus, to partition I_ω, the points where selections start and end must be determined. For each event in I_ω, one or more out of three possible conditions are true: (i) The event triggers that a new selection is opened. (ii) The event is part of open selections. (iii) The event triggers that one or more open selections are closed. To evaluate which condition is true, the splitter offers an interface that can be programmed according to the operator functionality and that provides the ability to store variables that capture internal state, e.g., about window start times. The interface comprises two predicates:

```
1: int nextStart = 0, slideTime, windowSize

2: bool Ps (Event e) begin
3:   if e.timestamp ≥ nextStart then
4:       nextStart = e.timestamp + slideTime - (e.timestamp % slideTime)
5:       return TRUE
6:   else
7:       return FALSE
8:   end if
9: end function

10: bool Pc (Event e, Selection s) begin
11:   if e.timestamp > (s.startTime + windowSize) then
12:       return TRUE
13:   else
14:       return FALSE
15:   end if
16: end function
```

Fig. 4. Predicates for the time sliding window operator.

```
1: bool Ps (Event e) begin
2:   if e.type == "A" then
3:       return TRUE
4:   else
5:       return FALSE
6:   end if
7: end function

8: bool Pc (Event e, Selection s) begin
9:   if e.type == "B" AND e.timestamp > s.startTime then
10:      return TRUE
11:  else
12:      return FALSE
13:  end if
14: end function
```

Fig. 5. Predicates for the sequence operator (A;B).

$P_s : e \rightarrow \text{BOOL}$, and

$P_c : (\sigma_{open}, e) \rightarrow \text{BOOL}$.

For each incoming event e, P_s is evaluated to determine whether e starts a selection, and P_c is evaluated with each open selection σ to determine whether e closes σ. Depending on the order of the evaluations of P_s and P_c on a newly arrived event, different semantics can be realized with respect to whether the start and close events are part of the selection or not.

Examples: In Figure 4, we showcase the predicates for a time-based sliding window operator [40]. Partitions are spanned over all events within a time window that moves by a sliding parameter for each new selection. In Figure 5, the predicates for a sequence operator Sequence(A;B) [13] of two events of type A and B are listed. Partitions are spanned between an event of type A and the next event of type B following it.

B. Runtime Environment

The runtime environment (RE) manages the execution of an operator instance. It receives the partitions assigned by the splitter and enforces that exactly those correlation steps are executed that correspond to the assigned partitions. In particular, when assigned partitions overlap, the RE prevents that the operator instance detects patterns that arise from the overlap, but enforces an isolation between the different partitions. This is done in the following way: When the operator instance processes an event that potentially starts a new correlation step, a function in the RE `isAssigned(startevent)` is called

TABLE I. SUPPORT OF CONSISTENT PARTITIONING. 'X' DENOTES SUPPORTED, '–' DENOTES NOT SUPPORTED.

	Ti	Tu	∨	;	∧	A	P
Pattern-sensitive	x	x	x	x	x	x	x
Run-based	–	x	x	–	x	–	–
Key-based	–	–	x	–	x	–	–

that signals whether it is a start event of an assigned partition or not. Further, when an event has been processed in all assigned partitions it is comprised in, the RE sends an acknowledgment to the splitter.

C. Expressiveness

To analyze the operators for which consistent stream partitioning is supported, we consider different categories of event processing operators from the literature. The following categories are considered: Sliding window operators (time-based (Ti) and tuple-based (Tu)) [40], disjunction (∨), sequence (;), conjunction (∧), aperiodic (A) and periodic (P) operators (all [13]) utilizing an unrestricted consumption mode [41], i.e., events can be used in several partitions without restrictions. Table I compares pattern-sensitive stream partitioning, the run-based approach proposed in [24] where event streams are partitioned into batches of a fixed size, and the key-based approaches like [26]–[29], [31], [32] where event streams are partitioned by a key contained in the events. Run-based partitioning cannot work consistently with operators for which the maximal amount of events comprised in a partition depends on the occurring events and thus is unknown before run-time, which is the case for (Ti), (;), (A) and (P) operators. Key-based partitioning is even less expressive. It does not allow for the consistent partitioning of event streams based on any other information than keys contained in every single event. Thus, the context of an event cannot be considered as it would be needed in (Ti), (Tu), (;), (A) and (P) operators.

Pattern-sensitive stream partitioning does not suffer from those limitations. For all analyzed operator types, the splitting logic is expressive enough to specify the partition start and end. The predicates of the time-based sliding window (Ti) operator are listed in Figure 4. For the tuple-based sliding window (Tu) operator, the predicates are similar: Instead of a time stamp, a counter is used or sequence numbers are compared to the partition start event. Disjunction (∨) and conjunction (∧) predicates check the corresponding conditions for opening and closing partitions, e.g., attribute values of an event. The sequence (;) predicates are listed in Figure 5. For aperiodic (A) and periodic (P) operators, arbitrary start and end conditions can be set, which can be composed of other operators. Accordingly, the predicates from the corresponding operators can be utilized to build the predicates of (A) and (P).

V. ADAPTING THE PARALLELIZATION DEGREE

In this section, we describe how to find and automatically adapt the parallelization degree at changing workloads. We aim for always deploying the optimal parallelization degree, which is the minimal degree that allows the operator to keep the assigned buffer limit with the required probability. To achieve this goal, we utilize *Queuing Theory (QT)* [42] to deduce a stationary distribution of the splitter queue length for

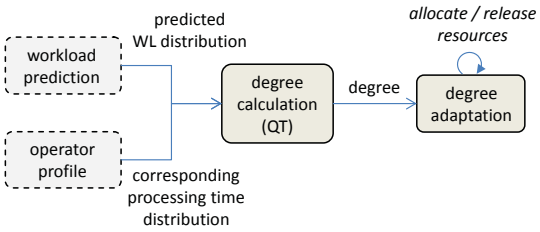


Fig. 6. Workflow of the adaptation of the parallelization degree.

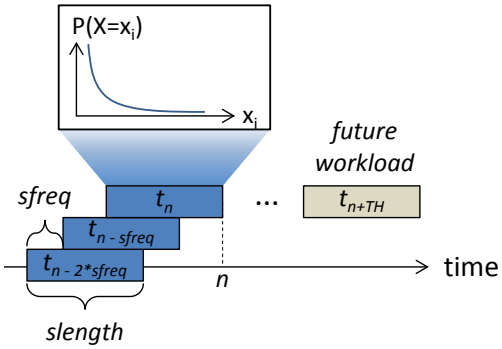


Fig. 7. Workload of an operator in time slices. The highlighted time slice shows an exponential distribution of inter-arrival times x.

a given parallelization degree. According to QT, we model the workload and processing time of an operator by probabilistic arrival and service processes of events. Corresponding to the arrival process, the events in I_ω determine the *workload* of ω. They are streamed from ω's predecessors, resulting in a process of inter-arrival times of events. Corresponding to the service process, the *processing time* of an instance of ω is described as the time it takes to process an event. This way, it is possible to dimension the operator solely based on the observed arrival and service processes without interference to the internal operator logic. Fig. 6 depicts the interplay of methods in our approach: Models of the predicted future workload and of the corresponding processing times are used for continuously calculating the optimal parallelization degree with QT methods. The calculated degree is established by an adaptation algorithm that allocates and deallocates operator instances.

A. Workload Monitoring and Prediction

A typical workload can consist of three parts [43]: Seasonal behavior, trends and noise. To account for short term fluctuations around an average value, i.e., noise, as well as medium or long term changes, i.e., seasonal behavior and trends, we divide the time scale into *time slices* (cf. Figure 7), denoted t_n the time slice that ends at a point in time n. To this end, we utilize a sliding time window with a length of $slength$ time units that for each new time slice moves by $sfreq$ time units. In each time slice, the inter-arrival time of arriving events follows a *probabilistic distribution*. Parameters of the distribution, for example the mean value, can change over subsequent time slices. In Figure 7, the distribution of inter-arrival times X in a time slice t_n is depicted, reflecting an exponential distribution. Note that depending on the distribution, all parameters of the distribution can change, e.g., the variance in a normal

distribution. To enable workload monitoring, the splitter logs the inter-arrival times of events.

To automatically match the logged data set of a time slice to a probabilistic distribution of inter-arrival times, we have developed a workload classification algorithm that comprises three basic steps [44]: The algorithm iterators over a set of supported distributions (exponential, deterministic, normal, uniform, Pareto, log-normal and Weibull distributions), the parameters of the selected distribution are estimated, and the goodness of the match with respect to the log is determined. The distribution that fits the logged data best is chosen.

To estimate the parameters of the selected distribution, the algorithm chooses between two approaches, depending on the selected distribution. When the parameters can be described as a function of the moments of the distribution function, the method of matching the *distribution moments* [44] is applied. This method is chosen when no higher moments are needed, i.e., when the selected distribution is the exponential, deterministic, normal, uniform or log-normal distribution. Otherwise, the *maximum likelihood method* [44] is applied. With this method, the parameters are calculated such that they would, when sampled from the selected distribution, lead to the given logged data with the highest probability.

To check the goodness of the selected distribution, statistical tests are applied [44], [45]. An important kind of goodness-of-fit test are tests that utilize statistics based on the *empirical distribution function (EDF)* [45]. The downside of this approach is that the necessary statistic tables are only available for standard distributions like the normal or the exponential distribution. In other more sophisticated cases, the algorithm utilizes the χ^2 *test for homogeneity*, where random samples from the selected distribution are created and checked against the monitored data [46]. This has the advantage that the hypothetic distribution is not directly included in the analysis, but only indirectly through the samples.

In order to predict the future workload TH time units ahead, we utilize methods from time series analysis on the workload monitored in the latest time slices. At highly dynamic workloads like Wikipedia page requests, state of the art forecasting methods yielded a mean relative error of about 20 % [43]. Such errors have to be taken into account by adding an overestimation factor when the workload shows characteristics that make it hard to yield accurate predictions. Depending on the distribution that has been monitored, a different number of parameters will be predicted. For instance, when considering an exponential distribution, only the mean value is relevant, while in a normal distribution also the variance is subject to prediction.

B. Operator Profiling

The processing time of an instance often depends on the workload: For example, in an operator aggregating values in a time-based window, higher event rates mean that the processed windows contain more events, so that the processing time in the window increases. To account for that, ω is profiled before run-time for different workloads on the target system. To create an *operator profile*, first of all an operator instance is instrumented. That means that there is added a monitoring function that measures how long it takes to process an event. The operator instance is then executed with different workload distributions that are expected to occur. From the measurements, the corresponding distributions of processing times are gained with the same distribution fitting methods as we use in workload monitoring.

```
1: function calculate_degree () begin
2:   c = current_degree                                  ▷ parallelization degree
3:   while true do
4:     P = ∑_{n=1}^{BL_ω} P(Q(t) = n)                    ▷ applying QT formulas
5:     if P < P_required then
6:       c = c + 1
7:     else
8:       if P ≥ P_required AND last_P < P_required then
9:         return c
10:      else
11:        c = c − 1
12:      end if
13:    end if
14:    last_P = P
15:  end while
16: end function
```

Fig. 8. Algorithm for calculating the optimal parallelization degree.

C. Degree Calculation

Recap that the optimal parallelization degree needs to be recalculated whenever the predicted workload changes. In this Section we present the algorithm for the parallelization degree calculation, which is listed in Figure 8. Given the predicted workload at time slice t_{n+TH} and the operator profile, the algorithm first calculates the probability that the queue length will be less or equal to BL_ω for a fixed parallelization degree c. If the probability is too small, c is increased, until the minimal c is found that yields a probability higher than or equal to $P_{required}$ which is returned by the function.

Let Q = $\{$Q(t) : t $\geq$ 0$\}$ be the random process of the queue length Q(t) at time t at a fixed parallelization degree, workload distribution and operator profile. In the core, the algorithm utilizes methods from QT to calculate the queue length probability P(Q(t) $\leq$ BL_ω), i.e., the probability that there are not more than BL_ω buffered in the queue. Depending on the workload distribution, the corresponding processing time distribution and the number of operator instances, the distribution of Q(t) can settle down for $t \to \infty$ to a stationary distribution, so that P(Q(t) $\leq$ BL_ω) can be calculated. Closed-form formulas to calculate P(Q(t) $\leq$ BL_ω) are available for M/M/c [47] and M/D/c [48] queuing systems[3]. The algorithm uses the available mathematical formulas to compute the queue length probabilities P(Q(t) $\leq$ BL_ω) (line 4). c is adapted accordingly, until the minimal c is found that yields P(Q(t) $\leq$ BL_ω) $\geq$ $P_{required}$ (lines 5 – 14).

[3]We follow Kendall's notation, A/B/c, A = workload distribution, B = processing time distribution, c = number of services. M = exponential distribution / Markovian process, D = deterministic distribution / constant. The queue capacity is infinite and the queuing discipline is FIFO in all models used in this work. The actual queue capacity of the system is indeed limited by physical constraints which are, however, typically much higher than the required buffer limits, so that they don't have to be considered in the mathematical model.

```
1: function adapt_degree () begin
2:   new_degree = calculate_degree()
3:   dif = last_degree − new_degree
4:   if dif < 0 then
5:       cancel |dif| instances in TH time units
6:   end if
7:   if dif > 0 then
8:       Try to recall dif cancellations
9:       n = number of recalled cancellations
10:      Deploy dif − n new instances
11:  end if
12:  last_degree = new_degree
13: end function
```

Fig. 9. Algorithm for adapting the parallelization degree.

However, for many distributions Q(t) does not settle down to a stationary distribution, and in practice not all predicted workloads and profiled service time distributions are exponentially or deterministically distributed. In that case, they are *approximated* by an exponential or deterministic distribution such that the *cumulative distribution function* (cdf) of inter-arrival times is higher than the cdf of the actual distribution. That way, the approximation yields smaller inter-arrival times. When approximating the processing time distribution, it is done in the opposite way: The cdf of the approximation must be smaller to yield larger processing times. Note that it is necessary to have a well-fitting workload and service time classification in order to make sure the approximation really yields smaller inter-arrival times and higher processing times than the real distributions.

D. Adaptation

Adaptation of the *parallelization degree*. The algorithm for adapting the parallelization degree is listed in Figure 9. It is initiated every *sfreq* time units. The new optimal parallelization degree is calculated with the algorithm from Section V-C based on the predicted workload in TH time units. If there is a difference dif between the new degree and the last updated parallelization, the adaptation of the degree in TH time units is prepared. There are two cases: (i) $dif < 0$ and (ii) $dif > 0$. In case (i) (line 4), the cancellation of $|dif|$ instances in TH time units is registered (line 5). At the time of cancellation, the operator instances are unregistered in the splitter, so that they do not receive new selections. The instances finish processing the assigned partitions and then are shut down. In case (ii) (line 7), in order to reduce the number of deployment operations, it is first checked how many canceled instances are not shut down yet and can be reused. Those n instances are registered in the splitter (lines 8–9). The rest of the $dif - n$ instances are newly deployed and registered in the splitter as soon as they are live (line 10).

Adaptation of *slength* and *sfreq*. $slength$ must be long enough to capture a sufficient number of events that capture the workload distribution with high confidence. For instance, consider the mean value in an exponential distribution. In order to reach a confidence interval of 95 % that does not exceed $\delta = \pm 5\%$ of the sample mean, following an approximation in [49] $slength$ is chosen such that the required number of measurements $num_measurements$ is 1600, while for $\delta = \pm 10\%$ a value of $num_measurements = 400$ is sufficient. There is a trade-off between the time needed to collect enough measurements and the statistical significance of the derived workload model. If the number of measurements in a time slice is too low, the significance of the workload model and thereby also the workload prediction are affected negatively. However, it is evident that at a low arrival rate of events, it takes a long time until enough measurements are available to reason about the distribution with high confidence. At an average arrival rate of 0.5 events per second, it takes 3200 seconds, that is more than 53 minutes, in order to collect 1600 measurements needed for $\delta = \pm 5\%$.

```
1: function process_measurement (time_slice t, measurement m) begin
2:   if (m.timestamp ≥ t.start + max(slength))
3:       or (t.#measurements ≥ num_measurements) then
4:       t.COMPLETED( )
5:   else
6:       t.ADD_MEASUREMENT(m)
7:   end if
8: end function
```

Fig. 10. Processing measurements in time slices with the QT-DYN algorithm.

We propose the algorithm *QT-DYN* that dynamically adapts $slength$: A time slice ends at a maximal time limit $max(slength)$ or when $num_measurements$ measurements are available. For instance, we can set $max(slength)$ = 3 minutes and $num_measurements = 400$. In that case, a time slice ends either when 400 measurements are taken or after 3 minutes have passed. This algorithm allows for setting bounds in the trade-off between time and accuracy of capturing the workload distribution. The algorithm is embedded into the procedure of processing measurements (cf. Figure 10), checking the conditions when a measurement in a time slice is being processed. The function signals when the time slice is completed, so that the workload distribution can be computed.

In setting $sfreq$, it must be small enough so that changes in the workload are detected timely, but the higher it is the less computational overhead is caused. Another important factor is how fast the workload is expected to change and what is the impact on the parallelization degree. If it changes often and very abruptly and a small change has high impact on the parallelization degree, a lower value of $sfreq$ can be beneficial.

VI. Evaluation

In this section, we evaluate the proposed method to adapt the operator parallelization degree by analyzing the queue lengths and processing latencies at different workloads. We compare the QT-based approach to an approach that adapts the parallelization degree based on the average CPU utilization (cf. Section VI-B3), building on a traffic monitoring scenario. Furthermore, we analyze the effects of approximating given workload and service time distributions. Finally, we evaluate the performance of the splitter in partitioning the incoming event streams.

A. System Parameters

All experiments were performed on a computing cluster consisting of 6 physical hosts with 8 cores (Intel(R) Xeon(R) CPU E5620 @ 2.40GHz) and 24 GB memory that are connected by 10-Gigabit-Ethernet connections. Up to 4 operator instances are deployed on one host. We assume that it takes

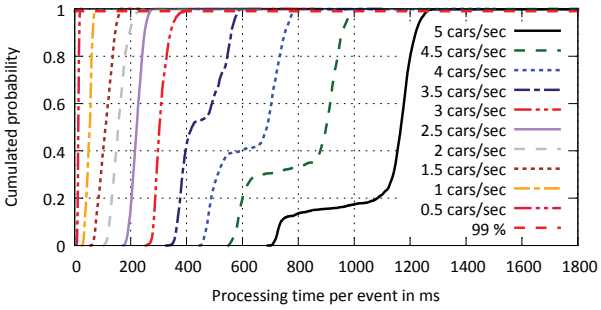


Fig. 11. Operator profiles.

60 seconds to deploy a new operator instance (TH = 60 seconds), which corresponds to worst case boot times, e.g., in the Amazon EC2 Cloud[4].

B. Dynamic Degree Adaptation

1) Traffic Monitoring Scenario: Recall the traffic monitoring scenario introduced in Section II-B. Given two sensors with synchronized clocks deployed at the beginning and end of a no-passing zone (`pos_1` and `pos_2`), the operator ω detects when a vehicle overtakes another one. We assume the following parameters of the setup: Between `pos_1` and `pos_2`, there is a distance of 15 kilometers. Following the speed limit, the vehicles will usually in average drive 60 kilometers per hour; however, there is a ratio of 10 percent of faster vehicles on the road which drive between 60 and 72 kilometers per hour (uniformly distributed). It has been shown in a number of measurement studies that the distribution of vehicles on a road is best modeled using a Poisson process [50], thus resulting in exponential distribution of inter-arrival times of vehicles at the checkpoints.

2) Operator Profiles: Fig. 11 shows the measured operator profiles with regard to our infrastructure. We only consider events of type `pos_2`, because events of type `pos_1` are just added to a list which is a negligible operation while events of type `pos_2` are compared to all events of type `pos_1` that have occurred in the selection, leading to noticeable processing times. The processing time grows polynomially with the event rate. For utilizing QT, we approximate the profile with the deterministic values of the 99-percentile and model the queuing system as an M/D/c queue.

Taking into account the operator profiles, the requirements on the queue length for the following evaluations are set to $BL_\omega = 15$ and $P_{required} = 95\%$, yielding low buffering delays of less than 4 seconds even at the highest traffic density, which allows, for instance, for direct feedback to a driver who violated the traffic rules. In our experiments, we choose $sfreq = slength$.

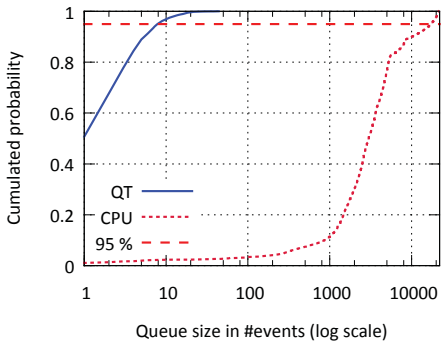
3) Fixed average inter-arrival time: Figure 12(a) shows the measured queue sizes of our QT-based approach, with a fixed $slength$ at 1600 events, compared to a reactive approach that adapts the parallelization degree depending on the average CPU load of active operator instances: Following Fernandez et al. [35], we add a new instance once the average CPU load over two subsequent time frames of 5 seconds is higher than 70 % and remove one instance when it is less than 50 %. In the QT approach, we used naive forecast [43], using the latest measured workload distribution as the predicted one. In the scenario, the inter-arrival time of cars has an average of 200 ms and the queue sizes are measured each 10 seconds over a run time of more than 8 hours. As can be seen in Figure 12(a), QT reaches a 95-percentile of 8 events, so that in 95 % of the time the queue length is 8 or lower and BL_ω is kept. The CPU based approach fails to keep BL_ω; the 95-percentile is at more than 17,000 events.

Furthermore, we measured the parallelization degree that is deployed in the different adaptation approaches (Figure 12(b)). In the QT-based approach, a stable degree of 8 is kept, as the workload classification reliably detects the workload distribution. The CPU-based approach however does not stabilize the degree. Instead, it is gradually increased to a certain level and then suddenly drops, taking another 30 minutes to increase it again. One reason is that the traffic monitoring operator works on large, overlapping partitions. The processing effort per event depends on the number of partitions that comprise the event as well as the number of events that already have been processed within a partition, as a new event from `pos_2` needs to be compared to a growing number of events from `pos_1` in a growing partition. Initially, the CPU based approach assigns many partitions to a small number of instances, because the CPU load in the beginning of processing a partition is still small. When the CPU load grows, it is already too late to add new instances, because the partitions that cause the overload are already assigned to the small number of instances. The algorithm adds more and more instances, and finally overshoots. Then, the same cycle starts again. This is an inherent problem of the CPU based approach in the scenario and similar results are observed for other CPU threshold values.

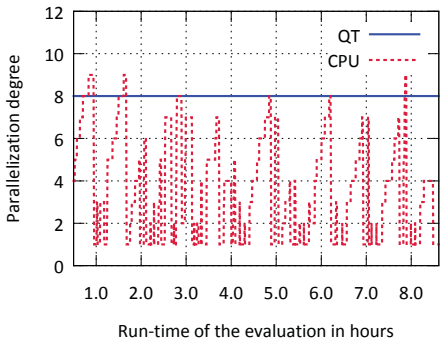
4) Dynamic average inter-arrival time: In the second scenario, the average inter-arrival time of events grows and shrinks over time to simulate a rush hour. The system is set up with an initial parallelization degree of 8. Within 2 hours, the traffic density grows from 0.5 cars per second to 5 cars per second, stays at that peak level for 2 hours and then gradually decreases over the next 2 hours back to 0.5 cars per second. All other assumptions and requirements are the same as in the previous scenario. When applying QT, we have evaluated different strategies for choosing $slength$: (i) Fixed sizes of the time slices containing $num_measurements$ measurements (denoted by QT-$num_measurements$), and (ii) the bounded dynamic algorithm QT-DYN (cf. Section V-D) with $max(slength)$ = 3 minutes and $num_measurements = 400$.

Figure 12(c) shows the cumulative distribution function (cdf) of the queue sizes for the QT-400, the QT-1600, the QT-DYN, and the CPU approach, where the 95-percentile is at 15, 18, 14 and 4,500 respectively. Looking into the logs of the QT-1600 approach, higher queue sizes than 15 mostly happen while scaling up, as it takes some time until all 1600 measurements for a time slice are available. Utilizing a smaller number of measurements in QT-400 and QT-DYN diminishes the problem while still giving a good estimation of the workload distribution. In Figure 12(d), the parallelization

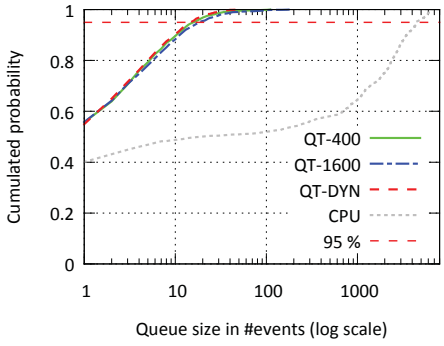
[4]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html



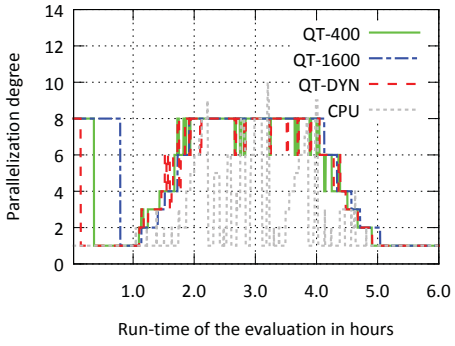
(a) Queue sizes of the QT and CPU approaches under static average inter-arrival time.



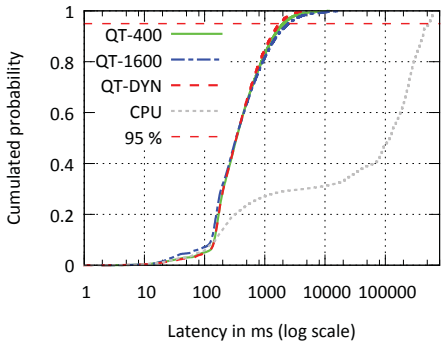
(b) Parallelization degree of the QT and CPU approaches under static average inter-arrival time.



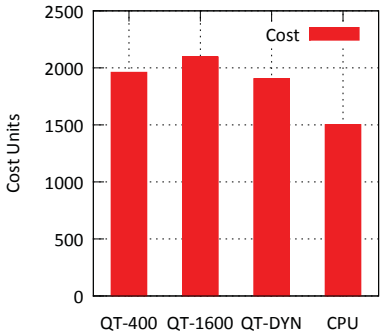
(c) Queue sizes of the QT and CPU approaches under dynamic average inter-arrival time.



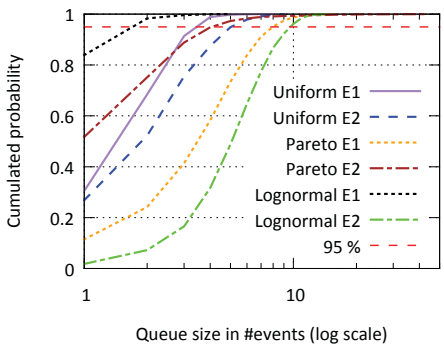
(d) Parallelization degree of the QT and CPU approaches under dynamic average inter-arrival time.



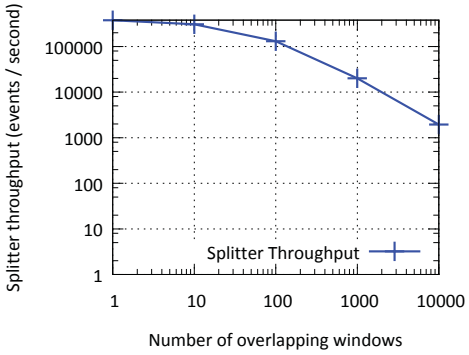
(e) Latency in the QT and CPU approaches under dynamic average inter-arrival time.



(f) Cost induced by the QT and CPU approaches under dynamic average inter-arrival time.



(g) Queue sizes at other distributions that are approximated.



(h) Splitter throughput in a tuple-based sliding window with different overlap.

Fig. 12. Evaluation results for different scenarios.

degree of the QT and CPU approaches is depicted. Similarly to the static scenario, we can observe in the QT approach a stable development of the degree that follows the workload while a heavily fluctuating degree occurs in the CPU approach. The dynamic algorithm QT-DYN shows a good adaptation behavior, as it allows for fast adaptations at a low event rate while increasing the statistical significance at a high event rate.

To further elaborate on the impact of the parallelization degree, we analyzed the imposed cost when running the dynamic scenario with the QT algorithms and the CPU based approach. We measure the cost for running operator instances based on a low pricing granularity of 1 minute, like it is given, for instance, in Google's cloud computing offers[5]. To keep a general pricing index, we assume that occupying a computing node for one minute costs 1 Cost Unit (CU). Figure 12(f) shows that in the QT approach, QT-DYN performs best with 1906 CU, followed by QT-400 with 1961 CU (+ 2.9 %) and QT-1600 with 2096 CU (+ 7.1 %). Furthermore, the CPU based approach does yield the lowest cost with 1502 CU. However, it fails to meet the buffer limit.

To elaborate on the link between the queue length and the timeliness of event detection, we have measured the *end-to-end latency* of the operator. That is the time between when an event that ends the detection of a pattern arrives at the splitter and the time when all corresponding events have been created and the event acknowledgment has arrived at the splitter. Figure 12(e) shows that the CPU based approach imposes a much higher latency than the QT based approach. In 5 % of the cases, the latency is even higher than 530 seconds and only in 29 % of the time a low latency of under 2 seconds is achieved. In contrast to this, the QT based approach yields a 95-percentile in latency of 1759 ms with QT-DYN, 2036 ms with QT-400 (+ 15.7 %), and 2471 ms with QT-1600 (+ 40.5 %).

Analyzing the results, it is evident that QT-DYN yields a better overall performance than QT-400 and QT-1600. While the improvement of the buffering level in comparison to QT-400 might not be significant and could be interpreted as a statistical deviation, the improvements in cost and especially in latency are undisputed.

C. Approximating distributions

Recap that according to the considerations in Section V-C, workload and processing time distributions that do not yield a stationary queue length distribution in a queuing system are approximated by exponential or deterministic distributions. To analyze the effects of such approximations, we consider several distributions: (i) The Pareto distribution representing *heavy tailed, skewed* distributions, (ii) the uniform distribution representing *short tailed* distributions, and (iii) the log-normal distribution representing the multiplicative *product* of many independent, positive random variables . We run two experiments for each distribution:

E1: In the first experiment, the workload follows the approximated distribution while the service time follows an exponential distribution with an average value of 300 ms. In the experiment, the given workload is approximated by an exponential distribution, building an M/M/c queuing system.

E2: In the second experiment, the workload follows an exponential distribution with an average value of 66.7 ms while the service time follows the approximated distribution. In the experiment, the given service time is approximated by a deterministic distribution, building an M/D/c queuing system.

[5]https://cloud.google.com/compute/pricing

TABLE II. PARAMETERS, CORRESPONDING APPROXIMATIONS AND CALCULATED PARALLELIZATION DEGREES FOR THE APPROXIMATED DISTRIBUTIONS.

	Parameters	E1	λ^{-1}	c	E2	D	c
Uniform	a = 100, b = 200		0.0432	**10**		0.199	**4**
Pareto	$x_{min} = 0.05$, $k = 2$		0.0667	**6**		0.5	**10**
Log-normal	$\mu = -1.0$, $\sigma = 0.2$		0.1266	**3**		0.586	**12**

We have chosen different parameters for the distributions, which are all listed in Table II in the column "Parameters". In the E1 and E2 section in that table, the parameters of the corresponding approximations are listed, i.e. the average value "λ^{-1}" of the exponential approximation of the workload in E1 and the value "D" of the deterministic approximation of the service time in E2. In both experiments, "c" denotes the parallelization degree computed with QT formulas.

BL_ω is kept in all experiments (cf. Figure 12(g)), so that the approximations show good results.

D. Splitter Throughput

In the splitter, the predicate logic for typical operators consists of only a fixed number of parameter comparisons that are done for each event in P_s and P_c. What scales up the effort in evaluating the predicates is the number of concurrently open selections, as for each open selection, P_c is evaluated on each event. We analyze this effect by evaluating the sliding tuple-based window of 10,000 events with an increasing number of overlapping windows between 1 and 10,000. Figure 12(h) shows that the Splitter yields high throughput that degrades proportionally to the number of overlapping selections.

VII. CONCLUSION

CEP systems can support IoT applications in detecting situations in the surrounding world. In this article we have identified two important shortcomings on the way towards low latency event detection in CEP systems. First, the state of the art lacks consistent parallelization models for a huge class of operators. Second, state-of-the-art systems are not equipped with a method to adaptively determine the optimal parallelization degree at fluctuating workloads in order to guarantee a buffering limit is met.

To this end, the proposed pattern-sensitive stream partitioning method supports the consistent parallelization of a wide class of important CEP operators. Furthermore, the proposed degree adaptation method is able to meet probabilistic buffering limits at highly fluctuating workloads.

Future work will aim at developing a latency model for operators comprising also communication and processing delays

and tackling the dependencies between operators in operator graphs.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers and S. Bhowmik for their helpful comments. This work is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG), grant RO 1086/19-1 ("Parallel complex event processing to meet probabilistic latency bounds").

REFERENCES

[1] International Data Corporation (IDC). (2014, April) The digital universe of opportunities: Rich data and the increasing value of the internet of things. Http://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm.

[2] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010.

[3] D. Miorandi, S. Sicari, F. D. Pellegrini, and I. Chlamtac, "Internet of things: Vision, applications and research challenges," *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497 – 1516, 2012.

[4] Y. Simmhan, B. Cao, M. Giakkoupis, and V. K. Prasanna, "Adaptive rate stream processing for smart grid applications on clouds," in *Proceedings of the 2Nd International Workshop on Scientific Cloud Computing*, ser. ScienceCloud '11. New York, NY, USA: ACM, 2011, pp. 33–38.

[5] Z. Jerzak and H. Ziekow, "The debs 2014 grand challenge," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 266–269.

[6] R. C. Fernandez, M. Weidlich, P. Pietzuch, and A. Gal, "Scalable stateful stream processing for smart grids," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14. New York, NY, USA: ACM, 2014, pp. 276–281.

[7] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[8] A. Buchmann and B. Koldehofe, Eds., *IT-Information Technology*. Oldenbourg Verlag, October 2009, vol. 51, no. 2009.

[9] G. Cugola and A. Margara, "Tesla: a formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '10. New York, NY, USA: ACM, 2010, pp. 50–61.

[10] G. G. Koch, B. Koldehofe, and K. Rothermel, "Cordies: Expressive event correlation in distributed systems," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '10, 2010, pp. 26–37.

[11] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[12] R. Mayer, B. Koldehofe, and K. Rothermel, "Meeting predictable buffer limits in the parallel execution of event processing operators," in *Proceedings of the IEEE International Conference on Big Data*, ser. BigData '14. IEEE, 2014, pp. 402–411.

[13] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data Knowl. Eng.*, vol. 14, no. 1, pp. 1–26, 1994.

[14] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik, "Scalable Distributed Stream Processing," in *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[15] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ser. DEBS '13. New York, NY, USA: ACM, 2013, pp. 27–38.

[16] B. Ottenwälder, R. Mayer, and B. Koldehofe, "Distributed complex event processing for mobile large-scale video applications," in *Proceedings of the Posters & Demos Session*, ser. Middleware Posters and Demos '14. New York, NY, USA: ACM, 2014, pp. 5–6.

[17] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran, "Mcep: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 6:1–6:24, Aug. 2014.

[18] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, "Migcep: Operator migration for mobility driven distributed complex event processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13, 2013, pp. 183–194.

[19] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, and U. Ramachandran, "Recep: Selection-based reuse for distributed complex event processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, 2014, pp. 59–70.

[20] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran, "Moving range queries in distributed complex event processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12, 2012, pp. 201–212.

[21] S. Chakravarthy and Q. Jiang, "Load shedding in data stream management systems," in *Stream Data Processing: A Quality of Service Perspective*, ser. Advances in Database Systems. Springer US, 2009, vol. 36, pp. 137–166.

[22] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch, "Overload management in data stream processing systems with latency guarantees," in *7th IEEE International Workshop on Feedback Computing (Feedback Computing'12)*, 2012.

[23] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: A second look at complex event processing architectures," in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, ser. GCE '11. New York, NY, USA: ACM, 2011, pp. 43–50.

[24] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul, "Rip: run-based intra-query parallelism for scalable complex event processing," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ser. DEBS '13, 2013, pp. 3–14.

[25] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using streammine3g," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, 2014, pp. 198–205.

[26] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10, 2010, pp. 21–21.

[27] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CloudCom '11, 2011, pp. 48–58.

[28] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10, 2010, pp. 119–130.

[29] (2014, September) Storm. Http://storm-project.net/.

[30] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10, Dec., pp. 170–177.

[31] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 59–72.

[32] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 53–64.

[33] P. Bhatotia, U. A. Acar, F. P. Junqueira, and R. Rodrigues, "Slider: Incremental sliding window analytics," in *Proceedings of the 15th International Middleware Conference*, ser. Middleware '14. New York, NY, USA: ACM, 2014, pp. 61–72.

[34] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, Mar. 2014.

[35] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 725–736.

[36] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1447–1463, June 2014.

[37] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13, 2013, pp. 15–26.

[38] S. Kounev, "Performance modeling and evaluation of distributed component-based systems using queueing petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.

[39] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate latency estimation in a distributed event processing system," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11, 2011, pp. 255–266.

[40] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[41] R. Adaikkalavan and S. Chakravarthy, "Seamless event and data stream processing: Reconciling windows and consumption modes," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I*, ser. DASFAA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 341–356.

[42] D. Gross, J. Shortle, J. Thompson, and C. Harris, *Fundamentals of Queueing Theory*, ser. Wiley Series in Probability and Statistics. Wiley, 2011.

[43] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[44] D. G. Feitelson, "Workload modeling for computer systems performance evaluation," book draft, to be published by Cambridge University Press in 2015. Version 1.0.1, typeset on April 8, 2014.

[45] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.

[46] T. M. Franke, T. Ho, and C. A. Christie, "The chi-square test: Often used and more often misinterpreted," *American Journal of Evaluation*, vol. 33, no. 3, pp. 448–458, 2012.

[47] S. K. Bose. (2014) M/m/m/inf queue. Http://nptel.ac.in/courses/117103017/6.

[48] H. Tijms, "New and old results for the m/d/c queue," *AEU - International Journal of Electronics and Communications*, vol. 60, no. 2, pp. 125 – 130, 2006.

[49] V. Guerriero, "Power law distribution: Method of multi-scale inferential statistics," *Journal of Modern Mathematics Frontier (JMMF)*, vol. 1, pp. 21–28, 2012.

[50] R. Mao and G. Mao, "Road traffic density estimation in vehicular networks," in *Proceedings of the 2013 IEEE Wireless Communications and Networking Conference*, ser. WCNC '13, April 2013, pp. 4653–4658.