Real-Time Middleware for Cyber-Physical Event Processing

CHAO WANG, CHRISTOPHER GILL, and CHENYANG LU, Washington University in St. Louis

Cyber-physical systems (CPS) involve tight integration of cyber (computation) and physical domains, and both the effectiveness and correctness of a cyber-physical system application may rely on successful enforcement of constraints such as bounded latency and temporal validity subject to physical conditions. For many such systems (e.g., edge computing in the Industrial Internet of Things), it is desirable to enforce such constraints within a common middleware service (e.g., during event processing). In this article, we introduce CPEP, a new real-time middleware for cyber-physical event processing, with (1) extensible support for complex event processing operations, (2) execution prioritization and sharing, (3) enforcement of time consistency with load shedding, and (4) efficient memory management and concurrent data processing. We present the design, implementation, and empirical evaluation of CPEP and show that it can (1) support complex operations needed by many applications, (2) schedule data processing according to consumers' priority levels, (3) enforce temporal validity, and (4) reduce processing delay and improve throughput of time-consistent events.

CCS Concepts: • **Computer systems organization** → **Embedded and cyber-physical systems**; **Real-time system architecture**; • **Software and its engineering** → **Middleware**;

Additional Key Words and Phrases: Industrial Internet of Things

ACM Reference format:
Chao Wang, Christopher Gill, and Chenyang Lu. 2019. Real-Time Middleware for Cyber-Physical Event Processing. *ACM Trans. Cyber-Phys. Syst.* 3, 3, Article 29 (August 2019), 25 pages.
https://doi.org/10.1145/3218816

1 INTRODUCTION

Real-time event processing is essential for cyber-physical systems (CPS), such as Industrial Internet of Things [8, 9, 23] systems, which must perform operations on sensor data carried by events and must respond to stimuli with quick and correct actions (e.g., in milliseconds [21, 22]). For example, smart electric grid applications require latency to be less than 50ms, and processing operations are conducted near the edge of the network to the extent possible [22].

Multi-sensor fusion is required by many real applications, such as position estimation, obstacle detection, and object tracking [7, 29, 31, 33]. By synthesizing data supplied by different sensors, multi-sensor fusion offers subscribers a more cohesive and reliable assessment of the environment. Such processing is typically multi-stage. For example, data from sensors (event suppliers) is first

This research was supported in part by NSF grant 1329861 and ONR grant N000141612108.
Authors' address: C. Wang, National Taiwan Normal University, Computer Science and Information Engineering, Taipei City 116, Taiwan; email: cw@ntnu.edu.tw; C. Gill and C. Lu, Washington University in St. Louis, Computer Science and Engineering, St. Louis, MO, 63130, USA; emails: {cdgill, lu}@wustl.edu.

2378-962X/2019/08-ART29 $15.00
https://doi.org/10.1145/3218816

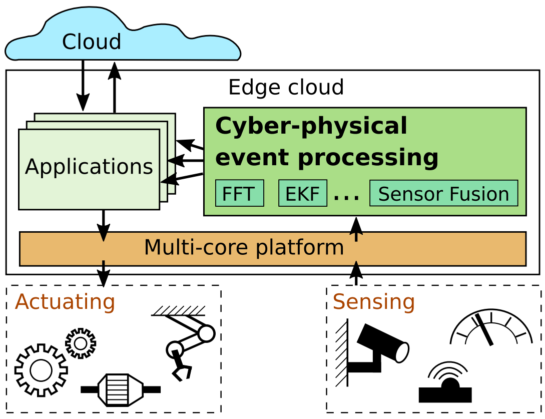


Fig. 1. Cyber-physical event processing.

passed through one or more filters for noise reduction, then a Fast Fourier Transform (FFT) is applied to the result to obtain frequency domain representations. Results from different processing streams are then combined, producing an event that represents a broader-spectrum assessment for applications (event consumers).

Real-time cyber-physical event processing must support configurable complex operations, meet applications' latency requirements, enforce temporal validity of events, and leverage multi-core platforms, as Figure 1 illustrates. First, applications often perform simple common operations (e.g., FFT) and complex operations that may be realized by combining other common operations (e.g., a multi-sensor fusion realized by filters, FFTs). Second, a cyber-physical system must accommodate applications' different latency requirements, and should allow applications to share processing and data. Duplicating complex operations (or even portions of them) across application features wastes both communication bandwidth and computational resources, and re-implementing such operations for each application may unnecessarily increase software complexity and decrease software reliability. Third, cyber-physical applications are often subject to temporal validity constraints. For example, for automotive driving features such as adaptive cruise control, where data from sensors are fused to provide range estimates, the relevance of each sensor reading may decrease over time, and outdated data should be discarded. Finally, to better serve the needs of real-time *edge computing* [23], an event processing service must efficiently work with streams of events in terms of memory allocation and throughput.

To address these needs, in this article we introduce CPEP, a real-time middleware for cyber-physical event processing, with the following four features: (1)*configurable processing operations* integrate both simple and complex event processing; (2) *processing prioritization and sharing* ensure that higher-priority events are processed first and reduce the likelihood of starvation of lower-priority ones; (3) *enforcement of temporal validity and shedding* maintains temporal validity constraints, identifying and removing outdated data; and (4) *efficient concurrent processing* minimizes memory allocation for events and can scale up throughput with the number of CPU cores.

We implemented CPEP within TAO, a mature and widely used open source middleware [18, 30], by adding the preceding capabilities. We compared CPEP with the Apache Flink stream processing platform [1], and our empirical results show that CPEP can (through prioritization) better prevent higher-priority processing from incurring unnecessary delay, (through operation sharing) help reduce latency of lower-priority processing, and (through shedding) improve throughput of time-consistent events.

2 RELATED WORK

Cyber-physical event processing is an essential part of modern Industrial Internet-of-Things architectures [20], and in many use cases [9, 23] it is critical to minimize the time it takes to respond to stimuli. To this end, both messaging middleware (e.g., Kafka [25]) and the Data Distribution Service (DDS [15]) have been deployed. Kafka provides fault tolerance and load balancing for delivery of time-stamped log messages, and provides an interface for implementing message processing, but does not differentiate messages according to consumers' priority levels. DDS provides QoS options for data (event) delivery but does not process events. In contrast, CPEP both differentiates messages according to consumers' priority levels and processes events subject to time consistency.

Apache Flink is an open source stream processing framework featuring high throughput, low latency event processing and windowing, and fault tolerance [1]. The Flink framework accepts multiple event streams and performs stream transformations according to a plan. The results are new event streams, which in turn can be used for further transformations or be delivered to event subscribers. Flink supports a distributed runtime environment, where JobManagers (masters) receive the processing plans from clients and then distribute them to TaskManagers (workers) for execution. Windowing in Flink is either time driven (e.g., every 30 seconds) or event driven (e.g., every 100 events) and is typically used for event aggregation. Flink does not support absolute and relative time consistency enforcement, both of which are critical to real-time cyber-physical event processing, nor does it differentiate stream processing. In contrast, CPEP supports both types of time consistency enforcement and can prioritize stream processing.

Time consistency has been studied in real-time databases [32, 35], where *absolute time consistency* means that the datum being used by a transaction still carries a timely measurement, and *relative time consistency* means that the data being used by a transaction are updated within a specified time interval [32]. In our CPEP architecture, we extend the definition of time consistency for real-time cyber-physical event processing, and provide a design and implementation to enforce time consistency and to shed invalid work.

Time consistency is needed by many real-world applications. For example, a fire detection system may deploy a rule that triggers an alarm when both smoke and a high temperature occur within a certain time interval [3]; in modern automotive systems, conflicting commands sent to the same set of actuators (e.g., throttle actuator and/or brake actuator) within a certain time interval may cause unsafe interactions [6]. Juarez Dominguez et al. [6] also surveyed other feature interactions in embedded systems.

In social network analysis [16], the popularity of a post, as well as the size of the involved community, is determined by scores that decrease over time. Although social network analysis is typically conducted at the scale of seconds or even hours, in our CPEP middleware we enforce time consistency at scales as fine as milliseconds, a resolution required by many cyber-physical applications. The field of Complex Event Processing (CEP) [27] offers rich semantics for expressing stimuli using sets of events [4, 24]. GraphCEP [28] processes events for social network analysis, and implements timetables for updating the ranking of posts and comments according to the progress of time. GraphCEP maintains time consistency (at the timescale of hours and seconds) but does not share computation among processing streams. In contrast, CPEP maintains time consistency at the timescale of milliseconds and supports sharing of computation between processing streams.

3 CYBER-PHYSICAL EVENT-PROCESSING MODEL

Our event processing model consists of three kinds of components: suppliers, an event service, and consumers. Each supplier pushes *typed data items*, which we call *events*, to the event service; the event service processes the events according to a graph that defines the needed operations and their input/output events, as illustrated in Figure 2; and a consumer subscribes to the output events

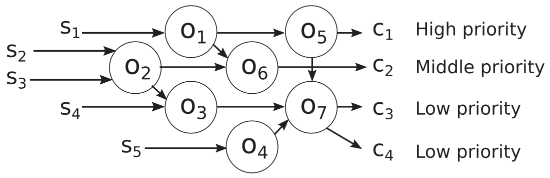


Fig. 2. Example graph of event processing streams in which s_i denotes a supplier; c_i, a consumer; and o_i, an operator.

of operations. Each supplier pushes events either periodically or sporadically. Each consumer is associated with a priority level. In practice, a supplier (consumer) may be mapped to a distinct sensor or other device, and multiple suppliers (consumers) may be mapped to a single device. The event service is executed within a single host.

In the following, we first define the processing within the event service, and then we define *absolute time consistency* and *relative time consistency* [32, 35],[1] which identify the temporal validity of an event or a set of events, respectively.

3.1 Event Processing

Event processing in our model is configured as a directed acyclic graph, as illustrated in Figure 2, and paths along the edges in the graph define the data processing streams for each consumer. The nodes of the graph are event processing operators, such as FFT, and the edges denote the precedence relations between operators. For example, Figure 2 shows processing streams for four consumers, and the streams for consumer c_2 involve operators o_1, o_2, and o_6. Operator o_1 has three downstream operators (o_5, o_6, and o_7), and operator o_6 has two upstream operators (o_1 and o_2). A complex operation, such as multi-sensor fusion, may be built from a set of common operators. Execution of an operator produces an event. We call events that are pushed from one operator to another *internal events*, the events pushed from suppliers *supplier events*, and the events pushed to consumers *consumer events*.

The event service schedules operators to process events. An operator is ready for execution if its specified dependencies are satisfied—for example, its upstream operators have completed processing, and all of its input events have arrived. The event service adds ready operators to the execution schedule. After execution, the same events will never be used again by the same operator. This ensures that cyber-physical operations, such as multi-sensor fusion, do not (prematurely) process newly arriving data in combination with previously used data.

In Section 4, we describe how CPEP first prioritizes operators based on the consumers' priority levels and then schedules the operators using a fixed-priority preemptive scheduling policy. We assume that the configuration of processing streams is specified by domain experts developing a particular application.

3.2 Absolute Time Consistency

Event e_i is temporally valid at time t if t falls within the *absolute validity interval* of e_i, defined by

$$\mathrm{abs}(e_i) = [t_b(e_i), t_e(e_i)), \tag{1}$$

where $t_b(e_i)$ and $t_e(e_i)$ respectively define the beginning and the end of the interval. Because only supplier events are associated with physical phenomena, we define an internal event's absolute validity interval to be the maximum overlap of all e_i's upstream supplier events' absolute validity

[1]We extend the definitions to make them suitable for real-time cyber-physical event processing, as described in Sections 3.2 and 3.3, respectively.

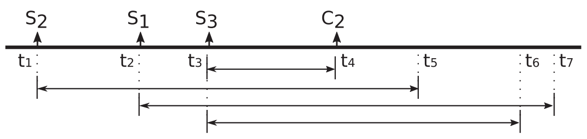


Fig. 3. Example timeline of event processing for consumer c_2 in Figure 2. Each vertical arrow marks either the event creation times at the suppliers or the event arrival time at the consumer.

intervals. Let $o(e_i)$ be the operator that produces e_i, and let I_{e_i} be the set of events required by operator $o(e_i)$. We have

$$t_b(e_i) = \max\{t_b(u) \mid u \in I_{e_i}\}; \tag{2}$$

$$t_e(e_i) = \min\{t_e(u) \mid u \in I_{e_i}\}. \tag{3}$$

If e_i is from a supplier, $t_b(e_i)$ is defined to be the creation time of the event. For example, as shown in Figure 3, $[t_1, t_5)$, $[t_2, t_7)$, and $[t_3, t_6)$ respectively represent the absolute validity intervals of the events from s_2, s_1, and s_3, and an event for consumer c_2 is temporally valid as long as it would arrive at c_2 before t_5. We assume that each supplier event's absolute validity interval is also specified by domain experts developing a particular application.

3.3 Relative Time Consistency

Here we reuse the definition of absolute time consistency. In general, we say that sets of events required by an operator may have relative time validity constraints, and each such constraint describes a mutually dependent timed relation between the events in a set. Formally, we say that I_{e_i} is temporally valid, if given sets $Q_j \subseteq I_{e_i}$, $j > 0$, we have

$$|t_b(e_x) - t_b(e_y)| \leq R_{Q_j} \tag{4}$$

for every two events e_x and e_y in Q_j. We call R_{Q_j} the *relative validity interval* of the set Q_j. From Equation (4), equivalently, I_{e_i} is temporally valid if

$$|t_b(e_p) - t_b(e_q)| \leq R_{Q_j} \tag{5}$$

for all Q_j, where event e_p is the earliest created and event e_q the latest created in each Q_j. Equation (5) offers an efficient way to verify the relative time consistency at runtime, which we will discuss in Section 4.3. As with absolute time consistency, we assume that set Q_j and interval R_{Q_j} are specified by domain experts.

4 CPEP DESIGN

CPEP processes cyber-physical events as follows. First, the graph of event processing streams is constructed from a configuration file, which specifies a list of the needed operators, with each item containing the operator type, the number of operators that immediately follow, and the indices to those operators. For each operator whose output event would be subscribed by a consumer, the operator is associated with a priority level mapped from the consumer's QoS specification. The event service assigns priority levels to the other operators by propagating upstream the priority levels of the consumer-facing operators, where each operator is assigned the highest priority level among its downstream operators. For example, the operators in Figure 2 would be partitioned into three priority groups (high: o_1, o_5; middle: o_2, o_6; and low: o_3, o_4, o_7). A supplier event's priority level is set to the highest priority level among the supplier-facing operators that would use it. With that priority assignment, the event service then reacts to the events pushed from suppliers,

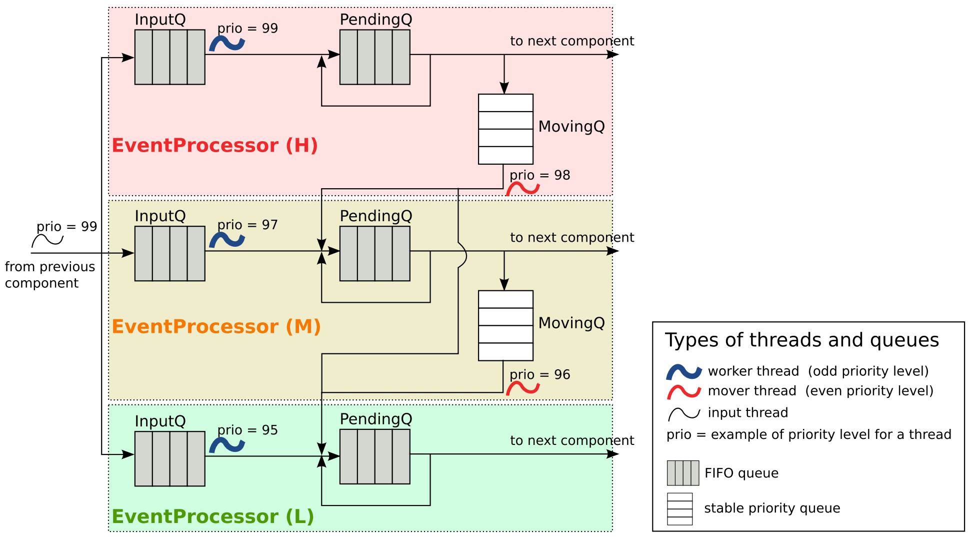


Fig. 4. CPEP EventProcessors for Figure 2 (H, high priority; M, middle priority; L, low priority).

processes them according to the graph of event processing streams, and pushes the resulting events to consumers.

4.1 Prioritized Processing and Sharing

The top-level component for processing is an *EventProcessor*, and there is one EventProcessor per priority level, as is illustrated in Figure 4. An EventProcessor includes two sets of active objects [26]: a set of *worker threads* in charge of processing same-priority events and a set of *mover threads* in charge of sharing the events that carry results of processing across priority levels (e.g., $o_1 \rightarrow o_6$, $o_2 \rightarrow o_3$, and $o_5 \rightarrow o_7$ in Figure 2).

A worker thread executes each operator that is *ready* due to arrival of a supplier event and/or completion of its upstream operator(s), and will proceed to process the next supplier event only when there remain no such pending operators. A mover thread shares the processing result in a tuple that contains both a reference to the pending operator and a reference to the resulting event.

CPEP prioritizes processing of streams and enforces the following two properties: (1) any processing of a certain priority level will preempt any cross-priority sharing from the same (or a lower) priority level, and (2) any cross-priority sharing from a certain priority level will preempt any processing of a lower priority level. This is achieved by assigning adjacent thread-level priorities to the worker and mover threads and scheduling them using a fixed-priority preemptive scheduling policy: starting from the EventProcessor of the highest priority level, we first assign all of its worker threads the highest thread-level priority and then assign all of its mover threads the next thread-level priority. We then repeat the process for the EventProcessor at the next priority level, using the remaining thread-level priorities. An example priority assignment is shown in Figure 4.

Each EventProcessor has three queues. The *InputQ* buffers all supplier events of the same priority level as that of the EventProcessor. The *PendingQ* holds the tuples for the subsequent same-priority operators along the graph of processing streams, and the *MovingQ* holds the tuples for cross-priority sharing. If cross-priority sharing is needed, the current worker thread puts the corresponding tuple into the MovingQ. An idle mover thread then moves the tuple from the MovingQ to the PendingQ(s) of the destination EventProcessor(s), which is then processed by the worker thread of each destination EventProcessor.

4.2 Concurrent Processing and Replacement

To improve throughput and reduce latency, on a multi-core platform CPEP can deploy multiple same-priority worker threads and execute independent operations concurrently (including independent portions of a complex operation), and different-priority worker threads can concurrently work using different CPU cores when possible. Concurrent processing is available in the following two circumstances: (1) when there are multiple event arrivals, be it from suppliers or from some preceding operators (e.g., via sharing) and (2) when an operator that is followed by multiple operators produces an event. In both cases, the PendingQ will be populated by multiple tuples. Concurrent processing is made possible in the following two ways: (1) *collaborative*, in which idle worker threads can take pending operators from the PendingQ after others have populated it (e.g., operators o_3 and o_4 in Figure 2 may be processed concurrently), and (2) *pipeline like*, in which CPEP allows a new series of processing along the graph of streams to start before the completion of the current series (e.g., processing for operators o_1 and o_2 in Figure 2 may start even before the completion of processing for operator o_6).

For each EventProcessor, we set both the number of worker threads and the number of mover threads to be equal to the number of CPU cores that are dedicated for processing but do not pin them to particular CPU cores. This can improve resource utilization and reduce processing latency.[2] For example, threads of lower priority levels, upon being preempted, can migrate to available CPU cores.

With concurrent and prioritized processing, for an operator requiring multiple events (e.g., operator o_6 in Figure 2), it is possible that an upstream operator may produce a second copy of a previously delivered event while the operator is waiting for an event from another operator upstream. In this case, the worker thread taking the second event will replace the previous event by it. Such replacement occurs each time a new event arrival is available, until the needed event from the other upstream operator is available. The replacement only takes effect on the immediate operator that receives the event, and other worker threads processing operators downstream from it will keep using the event they took when processing that operator.

CPEP can be configured to also enforce event replacement in the InputQ, in which case the buffer length is at most the number of different supplier event types, and each new event arrival of the same type will replace the previous event as long as the previous one has not yet been dequeued. Without event replacement, lower-priority InputQ's buffer needs to be large enough to accommodate preemption. The buffer length can be determined from workload profiling.

For both time and space efficiency, internally CPEP maintains a single instance for each event creation, and all workers may access the same instance concurrently if they need it. The event is stored in a centralized structure, named the *EventStore*, where each event is typed according to the supplier/operator that produced it. To accommodate pipeline-like concurrency, the EventStore includes one ring buffer per event type, and a new event of that type is put into the ring's next slot. The ring size is bounded by the maximum number of temporally valid instances of that event type at any given time point. For example, the ring size is equal to one if the event's absolute validity interval is smaller than the event's inter-arrival time. Slots are reclaimed in a lazy fashion, and only if there is a new event creation but no available slot. When needed, the slot holding the oldest event is reclaimed.

[2]See the related discussion for multi-processor global scheduling and partitioned scheduling [5].

4.3 Time Consistency Enforcement and Shedding

CPEP enforces both absolute time consistency and relative time consistency, and can be configured to have worker threads either mark or shed time-inconsistent events. With marking, CPEP simply labels such events and lets consumers decide what to do with them. With shedding, CPEP aborts any subsequent processing. Validation of both types of consistency is performed upon the invocation of each operator in the event processing graph. In addition, the absolute time consistency is also validated when the processing result is to be pushed to a consumer.

4.3.1 Absolute Time Consistency Validation. Given event e_i, to validate absolute time consistency, a worker thread compares the current time t against $t_e(e_i)$—for instance, the end time of the absolute validity interval. The worker thread reports a violation if $t > t_e(e_i)$. Upon a violation, if CPEP is configured to shed time-inconsistent events, the worker thread will update the value of $t_e(e_i)$ to the earliest end time among the absolute validity intervals of events on time-consistent upstream branches and will remove the event references of the time-inconsistent upstream branch.

4.3.2 Relative Time Consistency Validation. Let $S = \{e_1, e_2, \ldots, e_k\}$ be a set of event types subject to a specified relative validity interval, and following Equation (5) we say that the relative time consistency is violated if the maximum time difference of any two events in S is larger than the specified interval. To validate such consistency, for each operator CPEP maintains an ordered list of timestamps—one timestamp per event type in S. When a worker thread invokes an operator with a new event, it first updates the list and then compares the time interval between the last and the first element in the list against the relative time consistency interval. The worker thread reports a violation if the latter interval is smaller.

The preceding design ensures the correctness of enforcement: for an operator, it is necessary to keep track of the timestamp of each required event, because event replacements may occur before all of the needed events are available. For example, suppose that we only keep track of the earliest timestamp, say t_1, and the latest timestamp, say t_2, with respect to S. Given the second arrival of an event type where its previous arrival has defined t_1, the second arrival will define t_2, but t_1 will become undefined because its previous definition was from the same event type that now defines t_2.

4.4 Discussion on Distributed Settings

In the current version of CPEP, we describe a centralized service design where a single service host processes all events. Extension to support distributed settings deserves further study but is beyond the scope of this article. Here we discuss how the CPEP design may scale to support distributed settings. In particular, we focus on support for mapping the operators in the event processing graph onto multiple service hosts, to achieve distributed real-time cyber-physical event processing.

An ideal operator mapping would improve performance while preserving both latency differentiation and time consistency. In principle, first, in terms of latency, where to map the operators involves a tradeoff between inter-host communication delays and intra-host contention delays. Inter-host communication involves event transmission, and intra-host contention involves queueing and preemption. Therefore, for example, given an event processing graph with independent same-priority subgraphs, it may be advantageous to map operators of the same subgraph onto a single service host (thus minimizing inter-host communication) and operators of different subgraphs onto distinct service hosts (thus reducing intra-host contention). Further, it may be advantageous to map lower-priority operators onto multiple service hosts, provided that the amount of higher-priority interference on a service host outweighs the overhead of inter-host communication. Second, in terms of time consistency, events' validity intervals may drive mapping decisions

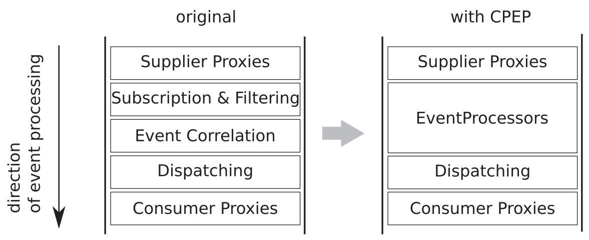


Fig. 5. Implementation within the TAO event channel.

as well. Third, in terms of memory efficiency, for multiple operators that share events, it may be advantageous to map them onto the same service host, to reduce additional event copies.

The operator mapping may be static, dynamic, or a hybrid. A static mapping may minimize the runtime overhead, with both the operators pre-allocated to each service host and the event routing pre-determined across service hosts. A downside is that the mapping may be pessimistic in the presence of aperiodic events. A dynamic mapping may reduce the pessimism at the cost of additional latency for runtime modules that must both decide where and how to route events and load operators if needed. A hybrid mapping also may be advantageous (e.g., adopting static mapping for periodic events and dynamic mapping for aperiodic events).

5 CPEP FRAMEWORK IMPLEMENTATION

In the architecture illustrated in Figure 4, we implemented MovingQ using C++11's standard priority queue, and to preserve the ordering of same-priority items we customized the priority queue's Compare type to use the timestamp taken at insertion as a tie-breaker. We implemented PendingQ using C++11's standard FIFO queue. For the configuration of event replacement, we implemented InputQ using C++11's standard FIFO queues to hold indices of each supplier event type. The indices are used to access a static storage for each supplier event type, and the size of the storage is equal to the number of different supplier event types. To enforce event replacement, each event arrival replaces the corresponding entry in the storage, and we push its index to the FIFO queue only if there is no such an index in the queue. For the configuration of no event replacement, we implemented InputQ using C++11's standard array as a ring buffer.

We protected all MovingQs, PendingQs, and InputQs with readers-writer locks to allow concurrent checks for non-emptiness of each queue. Each slot in the EventStore is also protected by a readers-writer lock to allow concurrent reads. To reduce priority inversion, we applied the pthread priority inheritance protocol to all worker threads and mover threads. At runtime, it takes $O(1)$ time to validate absolute time consistency, by comparing $t_e(e_i)$ against the current time. We maintain $t_e(e_i)$ by keeping track of the earliest end time for each upstream branch of $o(e_i)$.

The graph of event processing was implemented by an array of structs, each struct including both data structures for a node in the graph and pointers that build the graph's topology. The size of the struct was 464 bytes, including padding. The construction of the graph needs two linear scans through the array of structs: one scan for propagating priority-level information upstream and another scan for propagating time consistency information downstream. Notably, all constructions will complete before the system starts processing events.

We implemented CPEP within the TAO real-time event service [18]. Event suppliers and consumers in TAO are connected via one or more *event channels*, each containing five modules, as shown in Figure 5. Event filtering is conducted at both the Subscription & Filtering module and the Event Correlation module, where the former filters events according to event's type and source ID, and the latter filters events according to correlation rules defined over event types. The Dispatching

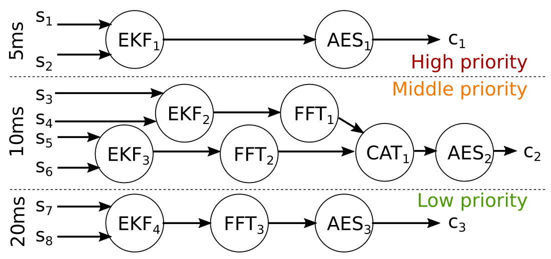


Fig. 6. Graph of event processing streams for Experiment Sets 1, 2, and 5.

module dispatches events to the subscribed consumers. Prior to our work, the TAO real-time event service only supports simple correlations (logical conjunction and disjunction) over events' headers, with non-sharing filters built per consumer. In contrast, CPEP provides prioritized event processing, enforces time consistency, and enables sharing of operations for better performance.

In our implementation, we kept the original interfaces of the Supplier Proxies and the Consumer Proxies so that suppliers and consumers can connect to the event channel as before. We replaced the Subscription & Filtering and Event Correlation modules with EventProcessors. We connected the Supplier Proxies to EventProcessors by a hook method within the push method of the Supplier Proxies module to put each event into the corresponding EventProcessor's InputQ. Worker threads dispatch their output events reactively.

6 EMPIRICAL EVALUATION

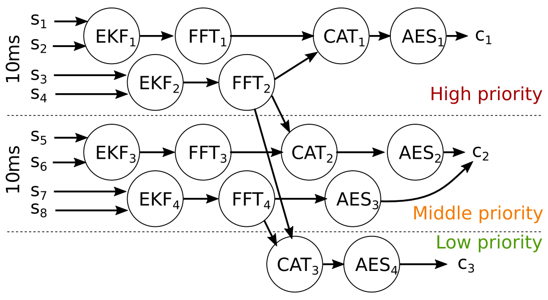
Here we present six sets of experimental results. In Set 1, we compare CPEP against Apache Flink in terms of latency, throughput, and memory footprint; in Sets 2 through 5, we evaluate the effectiveness of CPEP in terms of prioritization, sharing, absolute time consistency shedding, and relative time consistency shedding, respectively; and in Set 6, we present CPEP's overhead statistics. In all experiments, we enabled event replacement at InputQ; results of the configuration with no event replacement were reported in our previous work [34].

Our test-bed consists of three hosts: on Host 1, we ran all event suppliers (Pentium Dual-Core 3.2GHz, Ubuntu Linux with kernel v.3.19.0); on Host 2, we ran the CPEP event service and the Apache Flink server (Intel i5-4590 3.3GHz four-core machine, Ubuntu Linux with kernel v.4.2.0); and on Host 3, we ran all event consumers (Pentium Dual-Core 3.2GHz, Ubuntu Linux with kernel v.3.13.0). We connected the three hosts via a Gigabit switch running in a closed LAN. Host 2 had two NICs, and we used one for inbound traffic from Host 1 and another for outbound traffic to Host 3. Out of its four cores, on Host 2 we used three cores in the first experiment set (a comparison study with Apache Flink), and in the remaining experiment sets we offloaded the inbound network IO and CPEP's Supplier Proxies to the fourth core, and the three cores were dedicated to event processing.

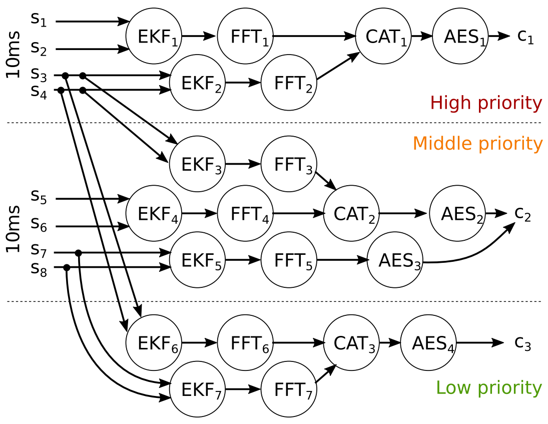
TAO's event channel was configured with the following parameters: we used the *default* factory, and we assigned *null* to `ECFiltering`, `ECSupplierFiltering`, `ECProxyConsumerLock`, and `ECProxySupplierLock`; *reactive* to `ECDispatching`, `ECConsumerControl`, and `ECSupplier-Control`; and *0* to both `ECConsumerControlPeriod` and `ECSupplierControlPeriod`.

We assigned real-time priority levels to both worker threads and mover threads, with the highest priority level set to 99. We also assigned 99 as the priority level of the thread for Supplier Proxies. We did not use the PREEMPT_RT patch [10].

Events were supplied at different rates, and consumers of events were assigned different priorities. We used two graphs of processing streams, shown in Figures 6 and 7, and in each case we



(a) With sharing operators.



(b) Without sharing operators.

Fig. 7. Graphs of event processing streams for Experiment Sets 3, 4, and 6.

evaluated the performance under different degrees of system workload. The workload we implemented demonstrates standard multi-sensor fusion operations [7, 29, 31, 33], with the following four operators combined in different ways: Extended Kalman Filter (EKF) [36], FFT [11], Concatenation (CAT) (implemented using C++'s `memcpy` function), and the Advanced Encryption Standard (AES) [14]. We ensured that the operators are thread safe by creating a distinct Kalman filter object per EKF operator and distinct FFTW matrices per FFT operator; the Libgcrypt library is by default thread safe, and we created a single Libgcrypt handler for all AES operators.

Because both absolute and relative time consistency enforcement use events' creation times to validate consistency, for event suppliers belonging to the same stream we chose to coordinate the phasing of event creations when evaluating absolute time consistency shedding (otherwise, the time difference between the creation times may dominate the latency and hence the shedding decision), and we chose not to coordinate event creations when evaluating relative time consistency shedding (otherwise, either all events would pass or all of them would be shed). Specifically, in the first four and the sixth experiment sets, we coordinated the event suppliers belonging to the same stream, and in the fifth experiment set, we chose not to coordinate as we did in the others. We coordinated the phasing of all suppliers that are upstream from a common consumer event and that have the same event rate, by dispatching them all from the same timer expiration rather than from individual independent timers.

We measured the end-to-end latency—that is, the time interval between the latest time a supplier pushed a required event and the time the consumer received the resulting event (e.g., $[t_3, t_4]$ in

Figure 3). We synchronized our test-bed's hosts via PTPd [13], an open source implementation of the IEEE Std. 1588-2008 Precision Time Protocol [19]. Both the service host's clock and the consumer host's clock were synchronized to the clock of the supplier host. The synchronization error was within 0.05ms.

In each of the first four experiment sets, we ran each sub-case 10 times and calculated the 95% confidence interval for each measurement; in the fifth experiment set, where we chose not to coordinate event suppliers, we ran each sub-case 40 times so that the 95% confidence interval converged. In each sub-case, we sequentially ran three phases: warm-up, measuring, and dumping. In the warm-up phase, we connected all event suppliers and consumers to CPEP, and had them start pushing and receiving events; in the measuring phase, we measured both the latency and the throughput of output events, and we kept all measurements in memory, which were then saved to disk in the dumping phase. For CPEP, the warm-up phase took 10 seconds. Apache Flink requires a longer warm-up time, and we set it to 75 seconds. In both cases, the measuring phase spanned 100 seconds.

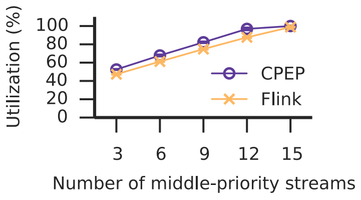
6.1 Experiment Set 1: Comparison with Apache Flink

Figure 6 shows the graph of event processing streams used in this experiment set. To cover different degrees of workload, we first deployed 3 copies of the high-priority streams, 3 copies of the middle-priority streams, and 12 copies of the low-priority streams, and then we increased the workload by deploying more copies of the middle-priority streams. Each supplier event carried a batch of one-byte datapoints[3]: each event supplied by s_1 and s_2 carried 512 datapoints (5ms event inter-arrival time); s_3 to s_6, 1,024 datapoints (10ms); and s_7 and s_8, 2,048 datapoints (20ms). Each output event for a high-priority consumer carried 512 one-byte datapoints; each output event for a middle-priority consumer carried 1,024 16-byte datapoints[4]; and each output event for a low-priority consumer carried 2,048 eight-byte datapoints.

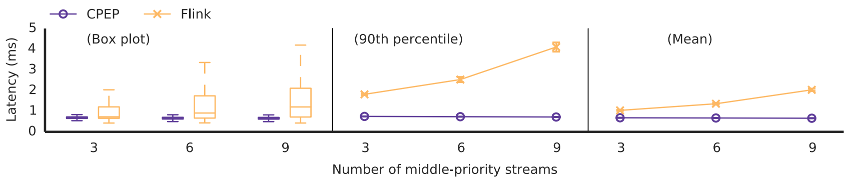
For a fair comparison to CPEP, in our Apache Flink (we simply call it *Flink* hereafter) Java application we used the same C++ libraries for the EKF, FFT, and AES operators, and invoked them via Java JNI. Considering that in this article we focus on single-host event processing, we configured Flink to run in a local environment, with a single Flink JobManager and TaskManager, respectively, and we ensured that they ran within the same JVM. To utilize multiple CPU cores, we set Flink's parallelism level equal to the number of CPU cores used for event processing (i.e., three) and also had the TaskManager use three task slots for concurrent execution. We implemented a socket data source to connect Flink with event suppliers and a socket data sink to connect it with event consumers. We chose not to use Flink's CEP library [12], because so far it does not support the removal of used events to avoid premature processing, which is required by cyber-physical event processing (see Section 3.1). We therefore implemented event processing using Flink's built-in data transformation functions. We also turned off checkpointing to reduce latency in Flink.

6.1.1 Latency Comparison. The latency results are shown in Figures 8 (high priority), 9 (middle priority), and 10 (low-priority) and Table 1; Figure 8(a) shows the CPU utilization normalized to the number of cores. The results show that CPEP outperformed Flink in terms of latency in all degrees of workload, and CPEP's latency performance followed the order of priority level. A major reason is that CPEP prioritizes event processing based on the consumer priority levels, whereas Flink

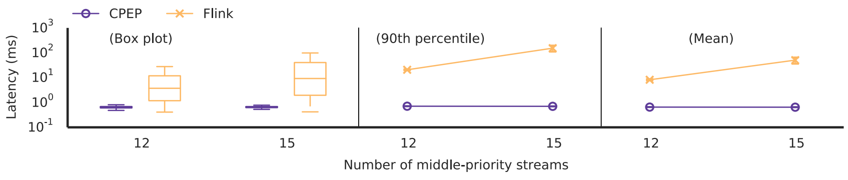
[3]For example, in structural health monitoring [17], FFT may require 2,048 samples to perform, and raw data may need to be transmitted to a base station if in situ processing is not sufficient.
[4]The FFT operator caused the increase in datapoint size, as the FFTW library transformed each byte of datapoints into an eight-byte real number (the single-precision version of FFTW).



(a) CPU utilization.



(b) Moderate workload (CPU utilization = 40%–80%).



(c) Heavy workload (CPU utilization > 80%).

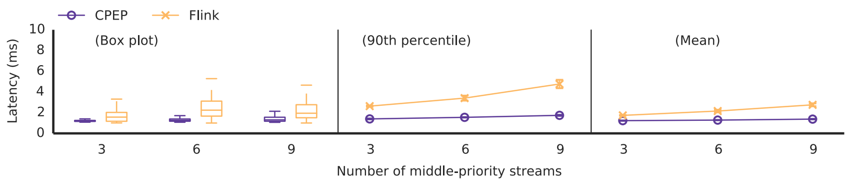
Fig. 8. Experiment Set 1: Latency results of high-priority streams.

Table 1. Experiment Set 1: The 99th Percentile Latency (ms)

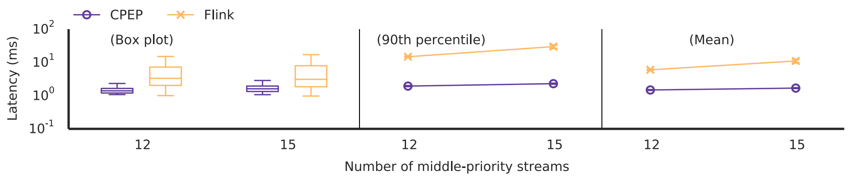
Priority	Service	Number of Middle-Priority Streams				
		3	6	9	12	15
High	Flink	3.8 ± 0.1	5.9 ± 0.2	12.6 ± 0.4	52.6 ± 4.1	448.9 ± 171.7
	CPEP	0.8 ± 0.0	0.7 ± 0.0	0.7 ± 0.0	0.7 ± 0.0	0.7 ± 0.0
Middle	Flink	4.5 ± 0.1	6.4 ± 0.2	11.3 ± 0.4	28.9 ± 0.5	107.9 ± 18.1
	CPEP	1.6 ± 0.0	1.8 ± 0.0	2.2 ± 0.0	2.5 ± 0.0	3.0 ± 0.1
Low	Flink	5.2 ± 0.3	7.4 ± 0.2	15.5 ± 0.6	43.3 ± 1.3	679.8 ± 274.0
	CPEP	3.7 ± 0.3	4.8 ± 0.2	6.8 ± 0.6	10.6 ± 1.0	33.4 ± 0.2

does not. In CPEP, tasks of higher priority will preempt lower-priority tasks' execution, whereas in Flink, tasks of higher priority may need to wait for the execution of tasks of lower priority.

Here we qualify the latency results by the corresponding inter-arrival times of events. In general, latency longer than the inter-arrival time implies that there are newer data available before a consumer has received the processed result. In practice, it is desirable to have latency even shorter than the inter-arrival time, and latency beyond the inter-arrival time may not be acceptable. Under moderate workload (CPU utilization = 40%–80%), both CPEP and Flink had 90th percentile latency and mean latency shorter than the corresponding inter-arrival time of events; under heavy workload (CPU utilization > 80%), however, the latency of Flink may exceed the inter-arrival time of

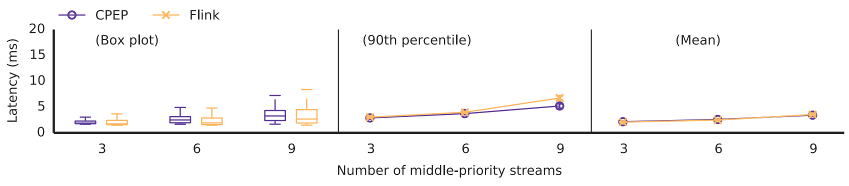


(a) Moderate workload (CPU utilization = 40%–80%).

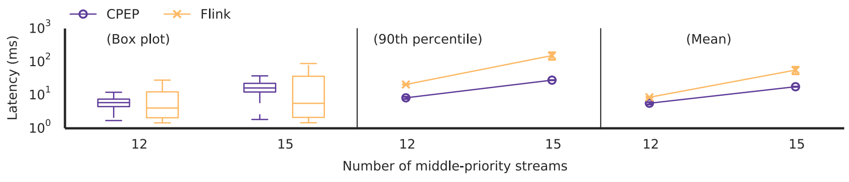


(b) Heavy workload (CPU utilization > 80%).

Fig. 9. Experiment Set 1: Latency results of middle-priority streams.



(a) Moderate workload (CPU utilization = 40%–80%).



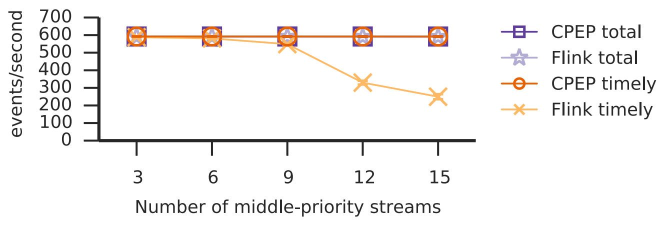
(b) Heavy workload (CPU utilization > 80%).

Fig. 10. Experiment Set 1: Latency results of low-priority streams.

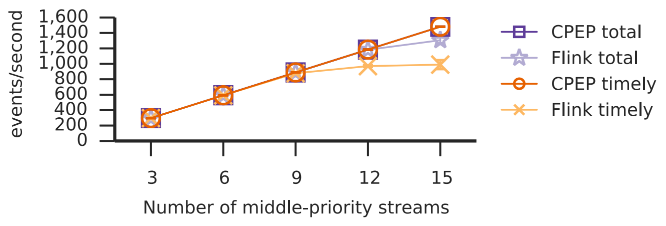
events. The high-priority streams, for example, had about 10ms mean latency (Figure 8(c)), which is twice the inter-arrival time.

For the 99th percentile latency, as shown in Table 1, CPEP's latency was shorter than the inter-arrival time in every sub-case except for that of low-priority streams along with 15 middle-priority streams (33.4 ± 0.2ms versus 20ms); in this sub-case, the CPUs were nearly saturated, and the latency of Flink was also beyond the inter-arrival time of events (679.8 ± 274.0 ms versus 20ms). For the low-priority streams, both the 90th percentile latency and mean latency of CPEP were similar to those of Flink, except for the aforementioned sub-case.

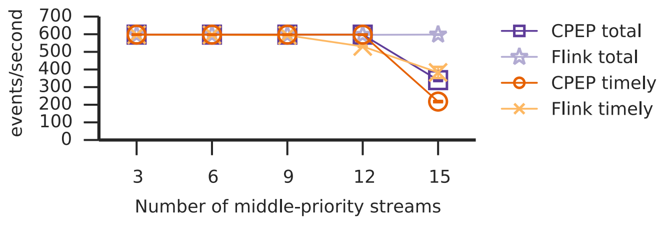
6.1.2 Throughput Comparison. The throughput results for each priority level are shown in Figure 11. The plots labeled by *CPEP/Flink total* show the throughput in terms of total events received by a consumer, and the plots labeled by *CPEP/Flink timely* show the throughput in terms of events received with latency shorter than the inter-arrival time. For the high-priority consumers



(a) High priority.



(b) Middle priority.



(c) Low priority.

Fig. 11. Experiment Set 1: Throughput results of each priority level.

(Figure 11(a)), both CPEP and Flink produced events at rates close to the event rates at the suppliers (three high-priority streams), but Flink produced much less timely events as we increased the workload. For example, in the presence of 12 middle-priority streams, Flink only produced about 300 events per second, half of the ideal rate. For the middle-priority consumers (Figure 11(b)), Flink started to produce fewer events as we increased the workload. For the low-priority consumers (Figure 11(c)) we made the same observation for CPEP. In general, CPEP and Flink behaved differently under heavy workload: in CPEP, the greater latency accrued to events of the lowest priority level, thanks to the prioritized processing; in Flink, the latency was distributed to all events, as Flink does not differentiate processing for different priority levels.

The prolonged latency also suggests that in terms of absolute time consistency, many events may not be considered valid, hence both wasting CPU resources and unnecessarily increasing delay. Indeed, many more violations of absolute time consistency can occur, as the latency here only accounted for the duration between the last event creation and the event delivery (e.g., $[t_3, t_4]$ in Figure 3), whereas the absolute time consistency accounts for the duration between the first event creation and the ultimate event delivery (e.g., $[t_1, t_4]$ in Figure 3). In Experiment Set 4, we evaluate CPEP's absolute time consistency enforcement and its shedding strategy. In the remaining five experiment sets, we do not use Flink and instead focus on evaluating CPEP's performance with different configurations, as CPEP clearly outperformed Flink in this comparison.

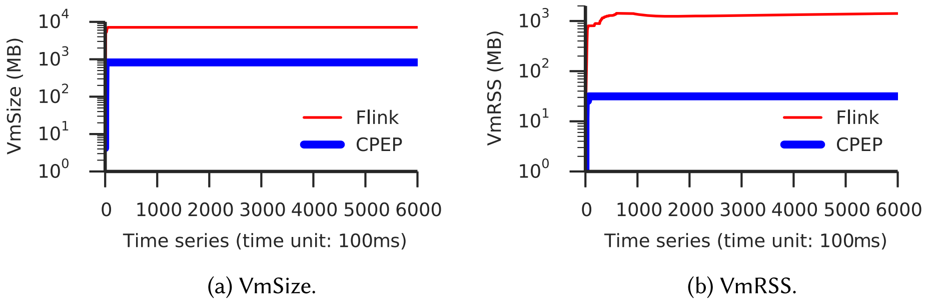


Fig. 12. Experiment Set 1: Memory footprint comparison.

6.1.3 Memory Footprint Comparison. We empirically compared the memory footprint of Flink and CPEP by querying the kernel data structures via `/proc/[pid]/status`. We show in Figure 12 the result of virtual memory size (VmSize) and the virtual memory resident size (VmRSS). Compared with Flink's memory usage, CPEP reduced more than 88% of the max VmSize (from 7146.5 to 824.6 MB) and more than 97% of the max VmRSS (from 1,424.8 to 31.5 MB). We also measured Java JNI's impact on the memory footprint, via the `pmap` utility. The sampling rate was 100ms, and we took 600 samples. The VmRSS of the JNI task was always 48KB, which is less than 0.005% of the total VmRSS.

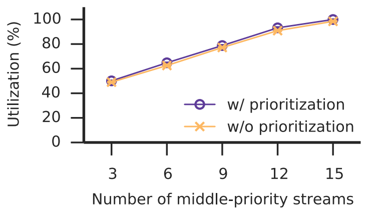
6.2 Experiment Set 2: CPEP Prioritization

CPEP can be configured for either prioritized or non-prioritized processing. In this experiment set, we compared the event latency of prioritized processing versus that of non-prioritized processing.

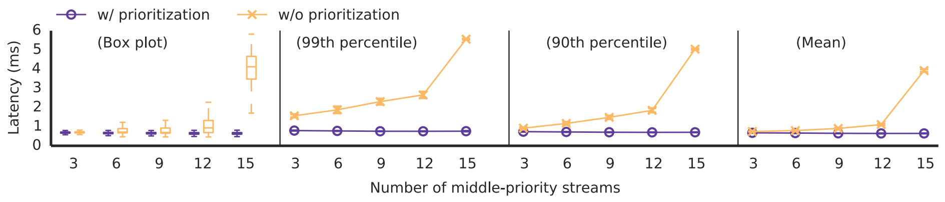
Figure 13 shows latency comparisons under different system workloads, with CPU utilization from around 45% to 95%, normalized to the number of cores used in processing event operations. As shown in Figure 13(b), prioritization maintained the latency of high-priority streams across different workloads. In contrast, without prioritization, the latency increased as the workload increased. Middle-priority streams exhibited similar behavior, shown in Figure 13(c). Low-priority streams exhibited the opposite behavior, shown in Figure 13(d), where prioritization led to higher latency for them than no prioritization did. This was because prioritization caused preemption of lower-priority processing. Nevertheless, the resulting latency was still less than half of the inter-arrival time of events (except for the sub-case of 15 middle-priority streams), and, as the next experiment set will show, sharing operations can further reduce the latency.

Figure 13 also shows that streams may have high tail latency even though the system was not heavily loaded (e.g., the 99th percentile latency of high-priority streams in the sub-case of six middle-priority streams, with the normalized CPU utilization 63% (Figure 13(b))). This happened because event arrivals of different streams were independent of each other and sometimes arrived close in time and contended heavily with each other. A stream may experience a higher tail latency under a higher workload, because in this case the processing of the stream is more likely to be delayed due to such contentions.

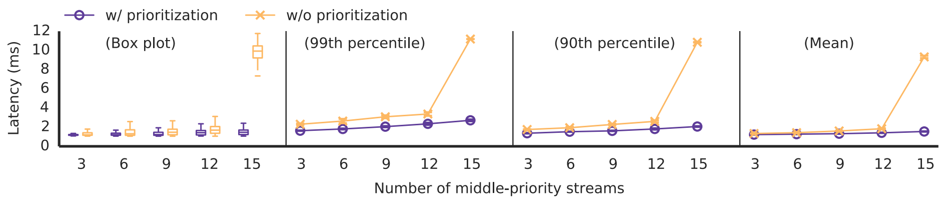
6.2.1 Results for Sporadic Events. We evaluated CPEP's performance with sporadic event streams by adding a random time offset in the range of ±2.5ms to the inter-arrival time of each event supplier. The result is shown in Figure 14. The performance is similar to the case without random time offsets (Figure 13), confirming that CPEP works for both periodic and sporadic event streams.



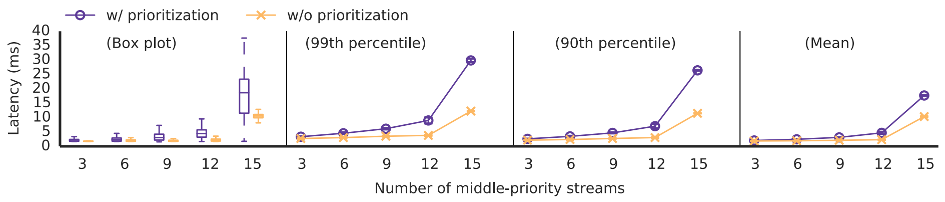
(a) CPU utilization.



(b) High priority.



(c) Middle priority.



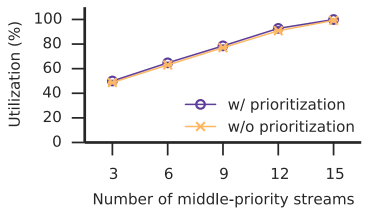
(d) Low priority.

Fig. 13. Experiment Set 2: Latency results.

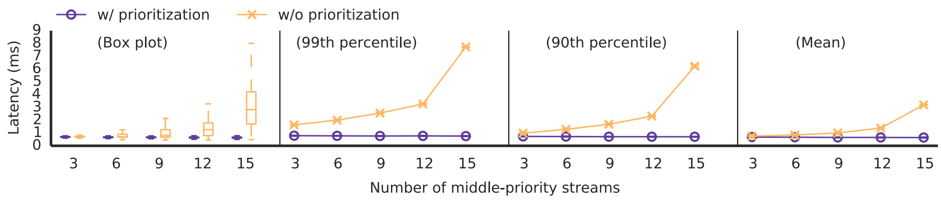
6.3 Experiment Set 3: Sharing Operators

In this experiment set, we evaluated the latency performance of sharing operations. We also enabled prioritized processing. We configured the graph of event processing streams as shown in Figure 7. Each event supplied by s_1 to s_4 carried 512 datapoints, and s_5 to s_8 carried 1,024 datapoints. The non-sharing version was constructed from the sharing version by duplicating the shared operators (FFT$_2$ and FFT$_4$) and all of their upstream operators (EKF$_2$ and EKF$_4$). All suppliers generated events with inter-arrival time of 10ms. Because the streams share operations, here we varied workload by deploying copies of the whole graph.

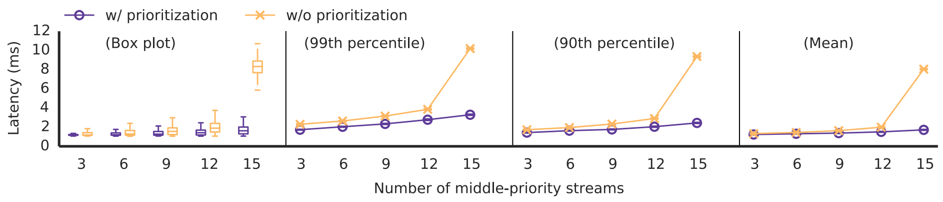
The latency results shown in Figure 15 confirm that sharing operations can help reduce event latency. In particular, lower-priority streams received higher reductions in latency, because the



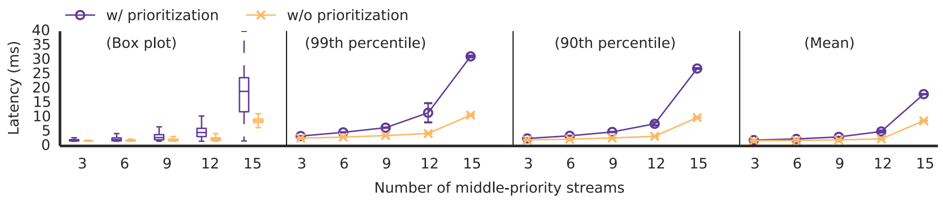
(a) CPU utilization.



(b) High priority.



(c) Middle priority.

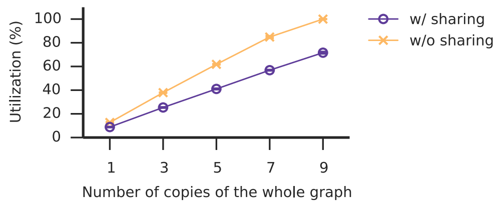


(d) Low priority.

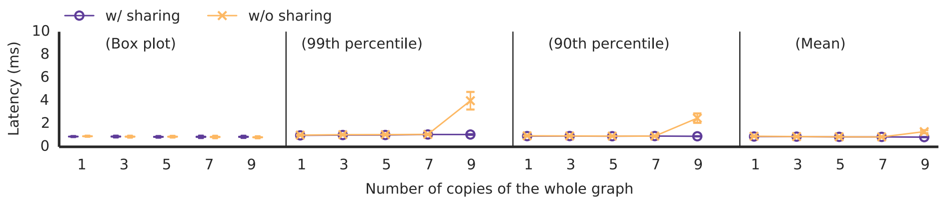
Fig. 14. Experiment Set 2: Latency results (sporadic events).

shared operations were done by the higher-priority counterpart. Figure 15(d) shows that as the workload increased, with sharing we may reduce the latency of lower-priority streams by more than 30% (e.g., the sub-case with seven copies of the whole graph), and with about the same mean latency, the system can accommodate 40% more streams (e.g., from five to seven copies of the whole graph). Besides reducing latency, sharing also saved CPU utilization. Figure 15(a) shows that with five copies of the whole graph, sharing can save about 20% in CPU utilization.

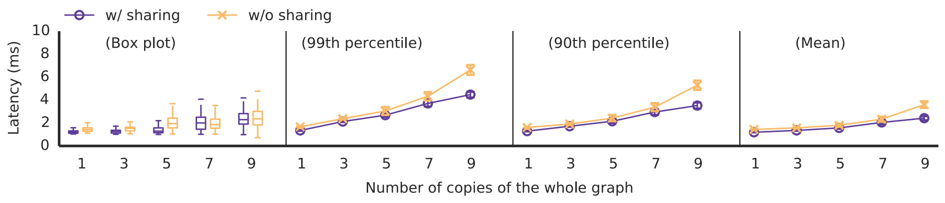
For lower-priority streams, the result in Figure 15(d) also suggests that the savings in processing time due to sharing may outweigh the time spent waiting for higher-priority streams. For higher-priority streams, as shown in Figures 15(b) and (c), sharing processed events to a lower-priority counterpart did not incur much overhead either.



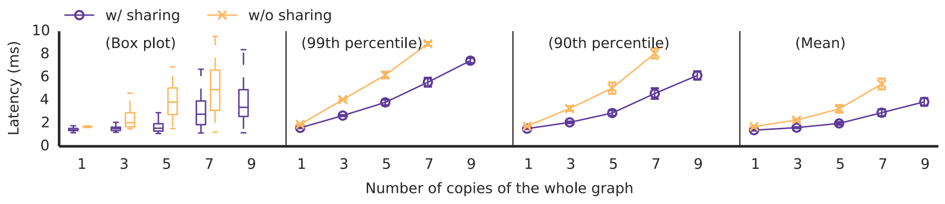
(a) CPU utilization.



(b) High priority.



(c) Middle priority.



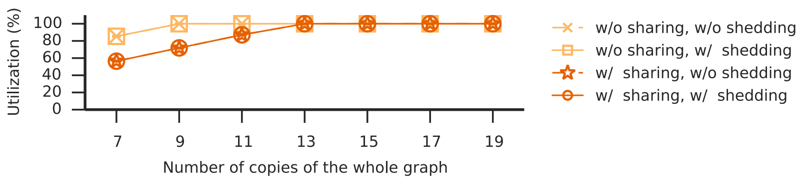
(d) Low priority.

Fig. 15. Experiment Set 3: Latency results. There was no output event of low priority for the case "w/o sharing" in the presence of nine copies of the whole graph.

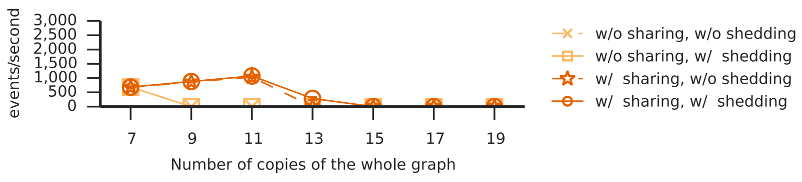
6.4 Experiment Set 4: Enforcing Absolute Time Consistency

In this experiment set, we evaluated the performance of absolute time consistency shedding and sharing, both separately and combined, in terms of *timely-throughput* (i.e., delivery of events within their absolute validity intervals). Here we reuse the graph of processing streams as shown in Figure 7. We set the absolute validity interval to 10ms.

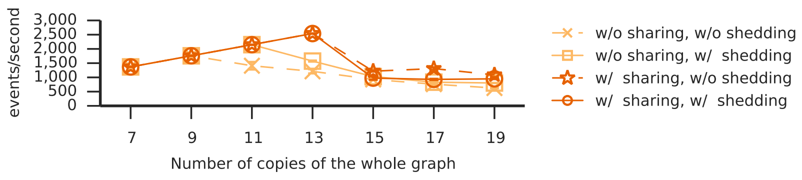
Figure 16 shows the result of timely-throughput in terms of events per second, where Figure 16(a) shows the CPU utilization. The benefit of shedding for timely-throughput was bounded by a range of system load, and for streams of higher priority the range shifted toward higher loads. This occurs because shedding did not take place under trivial loads, and because



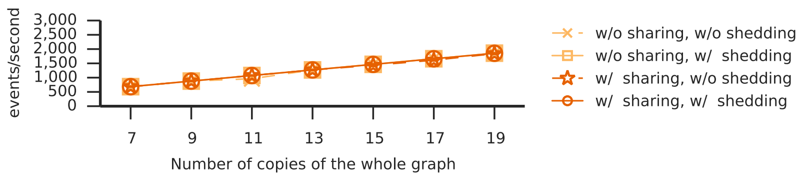
(a) CPU utilization.



(b) Low priority.



(c) Middle priority.



(d) High priority.

Fig. 16. Experiment Set 4: Timely throughput under different loads.

under heavy loads there is less slack time for streams to exploit. Considering that the absolute validity intervals approximate the supplier events' inter-arrival times, shedding occurred only when the total processing demand overloaded the CPUs. In particular, without sharing and with 11 copies of the graph of streams, the CPUs were saturated and shedding helped improve the timely-throughput of middle-priority streams (Figure 16(c)). With sharing and with 15 or more copies of the graph of streams, high-priority processing demand dominated CPU resources and therefore the timely-throughput of middle-priority streams dropped. Under such conditions, not shedding middle-priority demand at operators may improve timely-throughput, because the incurred delay may lead to event replacements in InputQ, which act essentially as early removals of events that, if otherwise dequeued, would still become outdated before processing completion.

High-priority streams (Figure 16(d)), although already protected by prioritization, still benefited from shedding, because when the system is overloaded shedding can reduce the amount of intra-priority contention—that is, the contention between the outdated processing and the processing

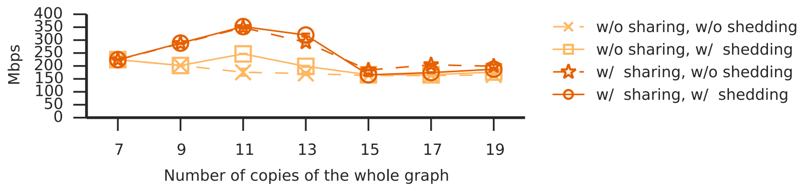


Fig. 17. Experiment Set 4: Total timely-throughput (Mbps).

that works to meet the timing constraint. Low-priority streams (Figure 16(b)) benefited most from sharing but may not benefit much from shedding, because (1) shedding works best when the system is under heavy load, in which case the lower-priority streams may already suffer non-trivial preemption, and (2) the benefit of shedding would accrue first to higher-priority streams due to prioritization.

Figure 17 shows the timely-throughput and includes all three priority levels. It shows that without sharing, even though the CPUs had been saturated in the presence of nine copies of the graph, with the help of shedding the system can keep producing at 200Mbps with up to 19 copies of the graph of streams; with sharing, we could still save about 20% CPU utilization in the presence of 11 copies of the graph.

6.5 Experiment Set 5: Enforcing Relative Time Consistency

In this experiment set, we used the graph of event processing streams as shown in Figure 6, and we did not coordinate the suppliers for each stream. We set the relative validity interval to 5ms for both high- and low-priority streams, and for the middle-priority streams we evaluated four relative validity intervals: 5.5ms, 7.0ms, 8.5ms, and 10.0ms. In all cases, we enabled prioritized processing and did not enable absolute validity shedding. In the following, we first show the performance with no relative validity shedding, and then we show the performance improvement by enabling relative validity shedding.

6.5.1 Effects of Violations of Relative Time Consistency. For the cases of no relative validity shedding, Figure 18(a) through (c) show the percentage of relatively time-consistent events produced, and Figure 18(d) shows the CPU utilization, for each case of the middle-priority relative validity interval. First of all, Figure 18(a) shows that the high-priority streams were protected by prioritized processing; they were not affected by different middle-priority validity intervals, nor were they affected by the increase in the number of middle-priority streams. Figure 18(c) shows that, also due to prioritized processing, the low-priority streams had a decrease in the percentage of relatively time-consistent events produced as we increased the number of middle-priority streams.

Figure 18(b) shows the results of middle-priority streams. The middle-priority streams are subject to more relative time consistency violations, because they depend on four event types, two more than the high-priority streams and middle-priority streams (see Figure 6). Figure 18(b) shows that with a longer relative validity interval, there was an increase in the percentage of relatively time-consistent event production. With the relative validity interval equal to 10ms, close to the event inter-arrival time, the percentage was close to 100%. In general, the percentage did not change much as we increased the number of middle-priority streams, because (1) the contention from high-priority streams did not change; (2) relative time consistency depends on the event creation time, which is indifferent to the time of processing; and (3) the CPUs did not saturate.

Figure 18(d) shows the CPU utilization normalized to the number of cores. With the result shown in Figure 18(b), this suggests that with no relative validity shedding, many more CPU cycles were wasted if we had a shorter relative validity interval.

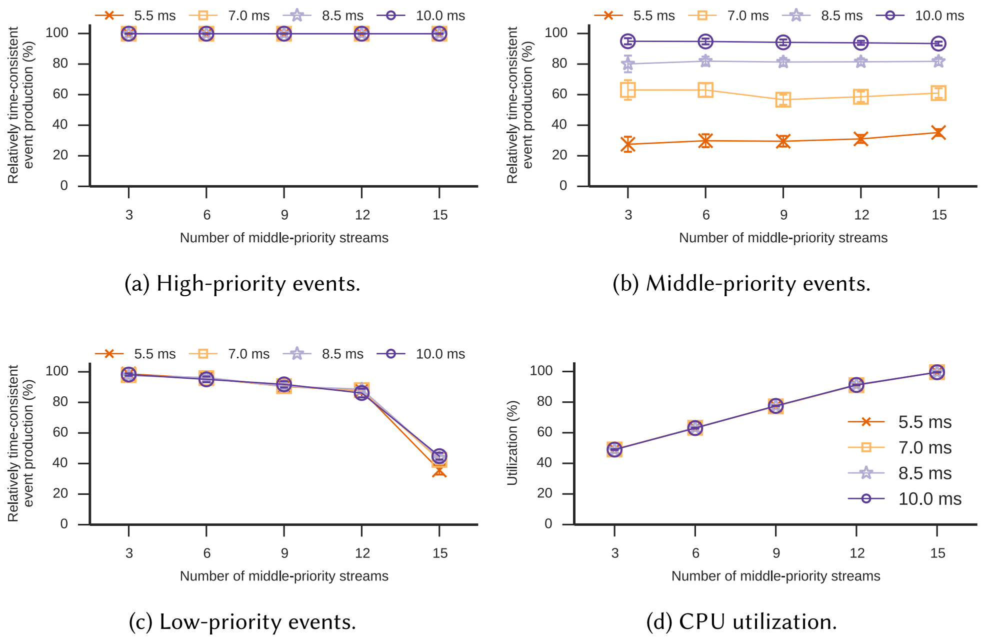


Fig. 18. Experiment Set 5: Percentage of relatively valid events with no relative validity shedding.

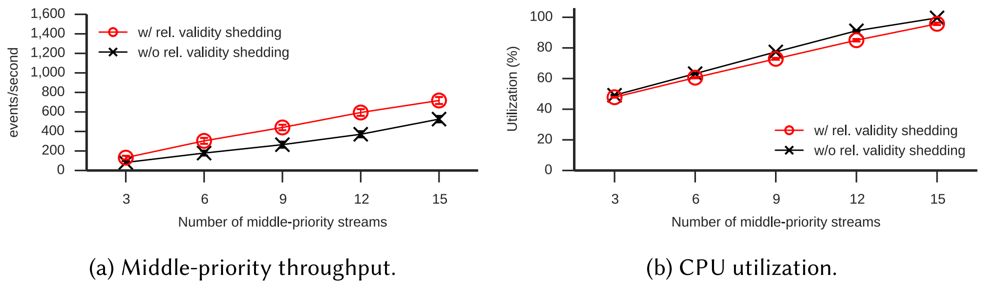
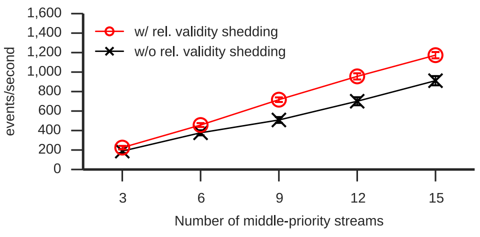
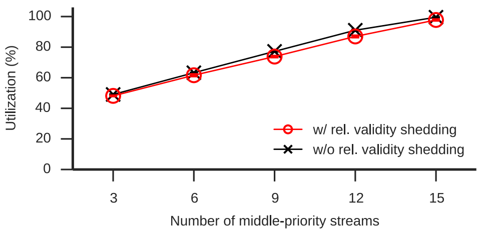


Fig. 19. Experiment Set 5: Relative validity shedding with relative validity interval = 5.5ms.

6.5.2 *Effects of Relative Time Consistency Enforcement and Shedding.* Figures 19 through 21 show the result of relative validity shedding with each relative validity interval, respectively. In all cases, relative validity shedding produced a higher relatively time-consistent throughput and at the same time reduced CPU utilization. With a shorter relative validity interval, there was more relative gain in throughput by shedding, because in that case there were more violations of relative time consistency and thus shedding saved more CPU cycles for relatively time-consistent processing. Figure 20, for example, shows the results for relative validity interval = 7.0ms, in which case shedding helped improve the throughput by about 200 events/second and saved about 5% in CPU utilization. It might not be possible to achieve the ideal throughput, which is $100 \times k$, where k is the number of middle-priority streams, because event suppliers were not coordinated and considering that without phase coordination the relevant events may spread out in time, violations of relative time consistency are more likely. With relative validity shedding, the throughput we had is much closer to ideal.

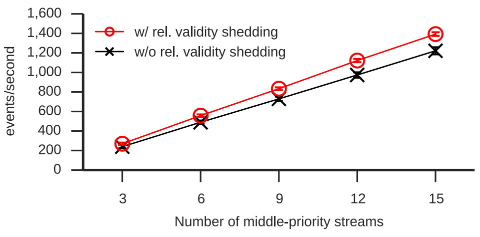


(a) Middle-priority throughput.

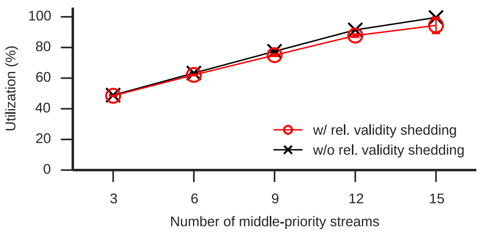


(b) CPU utilization.

Fig. 20. Experiment Set 5: Relative validity shedding with relative validity interval = 7.0ms.



(a) Middle-priority throughput.



(b) CPU utilization.

Fig. 21. Experiment Set 5: Relative validity shedding with relative validity interval = 8.5ms.

Table 2. Experiment Set 6: Latency of Event Queues

	InputQ	PendingQ	MovingQ
Enqueue	1.23ns	34.86ns	194.21ns
Dequeue	1.03ns	337.81ns	177.33ns

6.6 Experiment Set 6: Overhead Measurements

We evaluated CPEP's runtime overhead by showing both (1) queueing overhead and (2) preemption overhead. For queueing overhead, we measured both enqueue and dequeue times for the InputQ, PendingQ, and MovingQ. We took the average of 1 million measurements of each operation, and the result is shown in Table 2. The recorded dequeue time for the PendingQ included the time spent in binding operator's parameters to the dequeued metadata. The time complexity of the MovingQ is logarithmic in the queue length. Regarding the number of queueing operations per event, each supplier event needs at most one enqueue and one dequeue for the InputQ, each operator along the event processing graph needs exactly one enqueue and one dequeue for the PendingQ, and each transition between operators of different priority needs exactly one enqueue and one dequeue for the MovingQ. Our implementation permits concurrent checks for non-emptiness of a queue via readers-writer lock (see Section 5).

For the preemption overhead, we used `trace-cmd` to measure the number of scheduling events at runtime for 100 seconds. We compared CPEP's prioritization (using Linux's `SCHED_RR` real-time scheduler with real-time priorities) against a baseline version that disabled CPEP's prioritization (using Linux's default CFS scheduler with no real-time priority). Our result shows a 40% reduction in the number of context switches (the `sched_switch` events), with prioritization enabled than with it disabled, from about 60 switches per millisecond to about 37 switches per millisecond. As the SCHED_RR real-time scheduler permits global push and pull operations, it resulted in an increase in the number of thread migrations (the sched_migration events) from about 2.5 migrations per millisecond under CFS to about 6.3 migrations per millisecond under SCHED_RR. The preemption overhead therefore involves a tradeoff between a reduction in context switches and a smaller increase in the thread migrations. A detailed comparison between SCHED_RR and CFS can be found in the literature [2].

7 CONCLUDING REMARKS

In this article, we introduced the CPEP middleware for real-time cyber-physical event processing. CPEP features configurable operations, prioritization, time consistency enforcement, efficient memory management, and concurrent processing. We implemented CPEP within the TAO event service and empirically evaluated it in comparison to Apache Flink, showing that CPEP outperforms Flink in terms of latency, throughput, and memory footprint. Our further experiments showed that CPEP can both reduce latency and improve timely-throughput, through prioritization, sharing, absolute time consistency shedding, and relative time consistency shedding.

ACKNOWLEDGMENTS

The authors would like to thank reviewers for their feedback.

REFERENCES

[1] Apache. 2017. Apache Flink Home Page. Retrieved September 7, 2017 from https://flink.apache.org.
[2] Bjorn B. Brandenburg. 2011. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. Ph.D. Dissertation. University of North Carolina at Chapel Hill.
[3] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A formally defined event specification language. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*. 50–61.
[4] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys* 44, 3 (June 2012), Article 15, 62 pages.
[5] Robert I. Davis and Alan Burns. 2011. A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys* 43, 4 (2011), 35.
[6] Alma L. Juarez Dominguez. 2012. *Detection of Feature Interactions in Automotive Active Safety Features*. Ph.D. Dissertation. University of Waterloo.
[7] Nour-Eddin El Faouzi, Henry Leung, and Ajeesh Kurian. 2011. Data fusion in intelligent transportation systems: Progress and challenges—A survey. *Information Fusion* 12, 1 (2011), 4–10.
[8] Peter C. Evans and Marco Annunziata. 2012. Industrial Internet: Pushing the boundaries of minds and machines. *General Electric Reports.* Available at https://www.ge.com/docs/chapters/industrial_Internet.pdf.
[9] FogHorn. 2017. Industry Focus – FogHorn Systems. Retrieved September 18, 2017 from https://www.foghorn.io/industries/.
[10] The Linux Foundation. 2017. The Real Time Linux Collaborative Project. Retrieved September 18, 2017 from https://wiki.linuxfoundation.org/realtime/start.
[11] Matteo Frigo and Steven G. Johnson. 2017. FFTW Home Page. Retrieved September 18, 2017 from http://www.fftw.org.
[12] GitHub. 2017. Complex Event Processing for Flink. Retrieved September 7, 2017 from https://github.com/apache/flink/tree/master/flink-libraries/flink-cep.
[13] GitHub. 2017. PTP Daemon. Retrieved September 7, 2017 from https://github.com/ptpd/ptpd.
[14] GnuPG. 2017. The Libgcrypt Library. Retrieved September 18, 2017 from https://gnupg.org/software/libgcrypt.
[15] Object Management Group. 2015. Data Distribution Service (DDS). Retrieved September 18, 2017 from http://www.omg.org/spec/DDS/.
[16] Vincenzo Gulisano, Zbigniew Jerzak, Spyros Voulgaris, and Holger Ziekow. 2016. The DEBS 2016 grand challenge. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*. 289–292.
[17] Gregory Hackmann, Fei Sun, Nestor Castaneda, Chenyang Lu, and Shirley Dyke. 2008. A holistic approach to decentralized structural damage localization using wireless sensor networks. In *Proceedings of the 2008 Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 35–46.
[18] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt. 1997. The design and performance of a real-time CORBA event service. *ACM SIGPLAN Notices* 32, 10 (1997), 184–200.

[19] IEEE. 2008. IEEE standard for a precision clock synchronization protocol for networked measurement and control systems—Redline. *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)—Redline* (July 2008), 1–300.

[20] Industrial Internet Consortium. 2017. *Industrious Internet Reference Architecture*. Industrial Internet Consortium. Retrieved August 2, 2019 from https://www.iiconsortium.org/IIRA.htm.

[21] Real-Time Innovations. 2017. *Connext DDS at a Glance: Understanding the Software Framework That Connects the Industrial IoT. White Paper*. Real-Time Innovations.

[22] Kedar Khandeparkar, Krithi Ramamritham, and Rajeev Gupta. 2017. QoS-driven data processing algorithms for smart electric grids. *ACM Transactions on Cyber-Physical Systems* 1, 3 (March 2017), Article 14, 24 pages. DOI: https://doi.org/10.1145/3047410

[23] Daniel Kirsch. 2015. *The Value of Bringing Analytics to the Edge*. Hurwitz & Associates.

[24] Gerald G. Koch, Boris Koldehofe, and Kurt Rothermel. 2010. Cordies: Expressive event correlation in distributed systems. In *Proceedings of the 4th ACM International Conference on Distributed Event-Based Systems*. ACM, New York, NY, 26–37.

[25] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proceedings of the 6th International Workshop on Networking Meets Databases (NetDB'11)*. 1–7.

[26] Greg R. Lavender and Douglas C. Schmidt. 1995. Active object: An object behavioral pattern for concurrent programming. In *Proceedings of the Conference on Pattern Languages of Programs*.

[27] David C. Luckham. 2001. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison Wesley Longman, Boston, MA.

[28] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2016. GraphCEP: Real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*. ACM, New York, NY, 309–316.

[29] Daniel Piri. 2014. *Sensor Fusion for Nanopositioning*. Master's thesis. Vienna University of Technology, Austria.

[30] Douglas C. Schmidt. 2017. The ADAPTIVE Communication Environment (ACE). Retrieved September 18, 2017 from http://www.dre.vanderbilt.edu/~schmidt/ACE.html.

[31] Abu Sebastian and Angeliki Pantazi. 2012. Nanopositioning with multiple sensors: A case study in data storage. *IEEE Transactions on Control Systems Technology* 20, 2 (2012), 382–394.

[32] John A. Stankovic, Sang Hyuk Son, and Jörgen Hansson. 1999. Misconceptions about real-time databases. *Computer* 32, 6 (1999), 29–36.

[33] Ciza Thomas (Ed.). 2010. *Sensor Fusion and Its Applications*. Sciyo.

[34] Chao Wang, Christopher Gill, and Chenyang Lu. 2017. Real-time middleware for cyber-physical event processing. In *Proceedings of the 2017 IEEE/ACM 25th International Symposium on Quality of Service (IWQoS'17)*. IEEE, Los Alamitos, CA, 1–6.

[35] Ming Xiong, Rajendran Sivasankaran, John A. Stankovic, Krithi Ramamritham, and Don Towsley. 1996. Scheduling transactions with temporal constraints: Exploiting data semantics. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 240–251.

[36] Vincent Zalzal. 2008. KFilter—Free C++ Extended Kalman Filter Library. Retrieved September 18, 2017 from http://kalman.sourceforge.net.

Received September 2017; revised March 2018; accepted May 2018