Real-Time Middleware for Cyber-Physical Event Processing

Chao Wang, Christopher Gill, Chenyang Lu

Department of Computer Science and Engineering
Washington University in St. Louis
Email: {chaowang, cdgill, lu}@wustl.edu

***Abstract*—Cyber-physical applications are subject to temporal validity constraints, which must be enforced in addition to traditional QoS requirements such as bounded latency. For many such systems (e.g., automotive and 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 paper, we introduce CPEP, a new real-time middleware for cyber-physical event processing, with (1) extensible support for complex data processing operations, (2) execution prioritization and sharing, (3) enforcement of absolute 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' QoS requirements, (3) enforce temporal validity, and (4) reduce processing delay and improve throughput of temporally valid events.**

I. Introduction

Real-time event processing is essential for cyber-physical 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 or even microseconds [1]), in many applications such as those for automotive and Industrial Internet of Things [2]–[4] systems.

Multi-sensor fusion is used by many cyber-physical applications [5]–[7]. 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, a set of data from sensors (event suppliers) is first passed through one or more filters for noise reduction, and 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. First, applications often perform simple common operations (e.g., FFT) as well as complex operations that may be realized by assembling common operations (e.g, a multi-sensor fusion realized by filters, FFTs, etc.). 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 multiple application features wastes both communication bandwidth and computation 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 out-dated data should be discarded. Finally, to better serve the needs of real-time on-site computation in the Industrial Internet of Things, i.e., *edge computing* [3], an event processing service must efficiently work with streams of events in terms of memory allocation and throughput.

To address these needs, in this paper we introduce CPEP, a real-time middleware for cyber-physical event processing, with the following four features: *Configurable processing operations* integrate both common and complex data processing; *Processing prioritization and sharing* ensure that higher-priority ones get processed first and reduce the likelihood of starvation of lower-priority ones; *Enforcement of temporal validity and shedding* maintains temporal validity constraints, identifying and removing out-dated data; and *Efficient and concurrent processing* minimizes memory allocation for events and can scale up throughput with the number of CPUs.

We implemented CPEP within TAO, a mature and widely used open-source middleware [8], [9] by adding the capabilities mentioned above. Our empirical evaluations show that prioritized execution can save higher-priority processing from unnecessary delay, sharing of operations can help reduce latency of lower-priority processing, and shedding can improve throughput of temporally valid events.

II. 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

978-1-5386-2704-4/17/$31.00

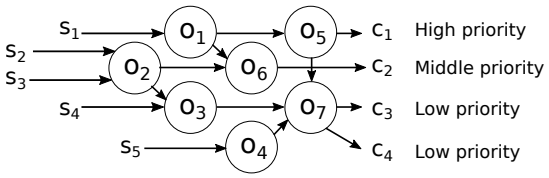


Fig. 1. An example graph of event processing streams. s_i denotes a supplier; c_i, a consumer; o_i, an operator.

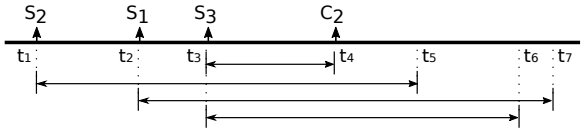


Fig. 2. An example timeline of event processing for c_2 in Fig. 1. Each vertical arrow marks either the event creation time at suppliers or the event arrival time at the consumer.

according to a graph that defines the needed operations and their input/output events, as illustrated in Fig. 1; a consumer subscribes to the output events 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.

A. Event Processing Streams

Event processing in our model is configured as a directed acyclic graph, as illustrated in Fig. 1, and paths along the edges in the graph define the data processing streams for each consumer. The nodes are event processing operators, such as FFT, and the edges denote the precedence relations between operators. For example, Fig. 1 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*, and the events pushed from suppliers and to consumers we call *external events*.

The event service schedules operators to process events. An operator is ready for execution if its specified dependencies are satisfied, e.g., its upstream operators have completed processing and all of its input events have arrived. The event service adds ready operators to the execution schedule. In Section III, we describe how CPEP first prioritizes operators based on the consumers' QoS parameters and then schedules the operators using a fixed-priority preemptive scheduling policy.

B. Time Consistency

Our model supports *absolute time consistency* [10][1], which identifies whether or not an event is temporally valid. Event e_i is temporally valid at time t if t falls within the *absolute validity interval* of e_i, defined by abs(e_i)=$[t_b(e_i), t_e(e_i))$. $t_b(e_i)$ and $t_e(e_i)$ respectively defines the beginning and the end of the interval. Because only external 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 external events' absolute validity intervals: Let $o(e_i)$ be the operator that produces e_i. $t_b(e_i) = \max\{t_b(u) \mid u \in$ input set of $o(e_i)\}$; $t_e(e_i) = \min\{t_e(u) \mid u \in$ input set of $o(e_i)\}$. If e_i is external, $t_b(e_i)$ is defined to be the creation time of the event. For example, as shown in Fig. 2, $[t_1, t_5)$, $[t_2, t_7)$, and $[t_3, t_6)$ respectively represents the absolute validity interval of 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, along with the configuration of processing streams, is specified by domain experts.

[1]Here we extend the definition to make it suitable for event processing.

III. CPEP Design

CPEP conducts cyber-physical event processing as follows. First, the graph of event processing streams is constructed from a configuration file, which specifies a list of the needed operators, each row containing the operator type, number of operators that immediately follow, and the indices to those operators. For each operator whose output event would be subscribed by a consumer, in the file the operator is associated with a priority level mapped from the consumer's QoS specification. Then the event service assigns priority levels to the other operators by propagating the priority levels of the consumer-facing operators upstream, where each operator is assigned the highest priority level among its downstream operators. For example, the operators in Fig. 1 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 the priority assignment, the event service then reacts to the events pushed from suppliers, processes them according to the graph of event processing streams, and pushes the resulting events to consumers.

A. Processing Prioritization and Sharing

The top-level component for processing is named an *EventProcessor*, and there is one EventProcessor per priority level, as illustrated in Fig. 3. An EventProcessor includes two sets of active objects [11]: a set of *Workers* in charge of executing operators, and a set of *Movers* in charge of sharing (across priority levels) the events that carry results of processing (e.g., $o_1 \rightarrow o_6$, $o_2 \rightarrow o_3$, and $o_5 \rightarrow o_7$ in Fig. 1).

A Worker works by executing each operator enabled either by the event or by completion of its operator(s), and will proceed to process a supplier event only when there remains no such pending operator. A Mover shares the processing result by passing (to the Worker of the destination priority level) a bundle that contains both a reference to the pending operator and a reference to the resulting event.

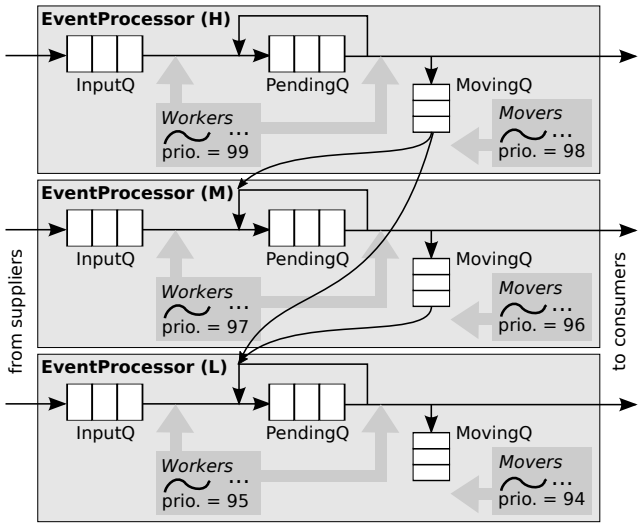


Fig. 3. The CPEP EventProcessors for Fig. 1 (H: high priority, M: middle priority, L: low priority). Abbreviation: prio. = thread-level priority.

The CPEP design 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; (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 Workers and Movers and scheduling them using a fixed priority preemptive scheduling policy: starting from the EventProcessor of the highest priority level, we first assign all its Workers the highest thread-level priority, and then assign all its Movers 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 Fig. 3.

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 bundles for the next same-priority operators along the graph of processing streams, and the *MovingQ* holds the bundles for cross-priority sharing. If cross-priority sharing is needed, the current Worker puts the corresponding bundle into the MovingQ. An idle Mover then moves the bundle from the MovingQ to the PendingQ(s) of the destination EventProcessor(s), which is then processed by the Workers of each destination EventProcessor.

B. Efficient and Concurrent Processing

By design, multiple Workers may work concurrently on independent portions of a complex operation to improve throughput and reduce latency. For each EventProcessor, we set the number of Workers equal to the number of CPU cores dedicated for processing, and concurrent processing is manifested in the following two ways: (1) *Collaborative*: when multiple operators are pending to be processed, each idle Worker selects one such operator and executes it. For example, in the graph shown in Fig. 1, o_3 and o_4 may be processed concurrently. (2) *Pipeline-like*: if the event's absolute validity interval is larger than the event's inter-arrival time, CPEP allows a new series of processing along the graph of streams to start before the completion of the current series. For example, in Fig. 1 processing for o_1 and o_2 may start even before the completion of processing for o_6.

For both time and space efficiency, CPEP maintains zero-copy semantics for each event creation. Each event is typed according to the supplier/operator that produces it, and once created, the event is stored in a centralized structure, named the *EventStore*, and Workers may access it concurrently when needed. To accommodate pipeline-like concurrency, the EventStore includes one ring buffer per event type, and the ring size is bounded by the maximum number of temporally valid instances of that event type at any given time point; the ring size is equal to one, for example, if the event's absolute validity interval is smaller than the event's inter-arrival time. Whenever a Worker takes a new supplier event or produces a new event, it puts the event into the ring's next slot.

By definition, an operator cannot be processed until all the required upstream events are available to it. Whenever a required event is available, CPEP binds the operator's local event reference to the corresponding slot in the EventStore. For pipeline-like concurrent processing, the same slot will be used for all downstream operators' local binding if they need that type of event, and a new event of that type will only be used in the new series of processing. An operator releases its local binding if the operator's processing has completed or if the event has violated its absolute time consistency.

C. Enforcement of Time Consistency and Shedding

Absolute time consistency is enforced when a Worker selects an operator or when it is ready to push the processing result to a consumer. The Worker compares the value of expiration time $t_e(e_i)$ of event e_i against the current time, t, and reports a violation if $t > t_e(e_i)$. CPEP can be configured to have Workers either mark or discard time-inconsistent events. With marking, the handling of such events is deferred to event consumers. With discarding, the corresponding downstream processing is aborted, resulting in load *shedding*.

Using shedding, in the presence of a violation of absolute time consistency, the Worker will release the related event-bindings. Let e_i be the event that accounts for the violation. The released event-bindings are those belonging to the same upstream branch of e_i. Then the Worker will set the value of $t_e(e_i)$ to the earliest end time among the absolute validity intervals of those events of temporally valid upstream branches.

IV. CPEP Framework Implementation

We implemented the MovingQ using C++11's standard priority queue; to keep 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 the PendingQ using C++11's standard FIFO queue. We implemented each InputQ using C++11's standard array and we protected it by a readers-writer lock to enable concurrent

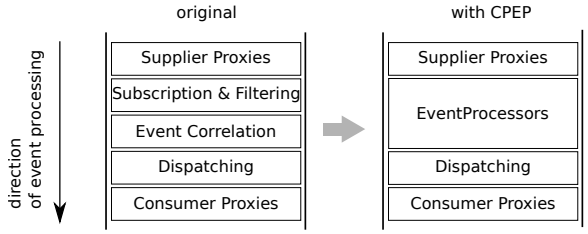


Fig. 4. Implementation within the TAO event channel.

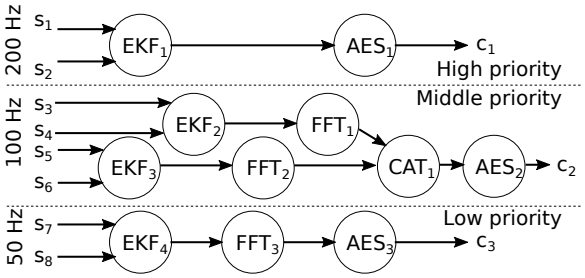


Fig. 5. Experiment 1: The graph of event processing streams.

reads. For the same reason each slot in the EventStore is also protected by a readers-writer lock. To reduce priority inversion, we applied the pthread priority inheritance protocol to all Worker threads and Mover threads. At run-time, 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 due time for each upstream branch of $o(e_i)$.

We implemented CPEP within the TAO real-time event service [9]. Event suppliers and consumers in TAO are connected via one or more *event channels*, each containing five modules, as shown in Fig. 4. 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 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 processing of data carried by events, 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 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.

V. Empirical Evaluation

We evaluated the effectiveness of CPEP in terms of prioritization, sharing, and shedding, with the following setup:

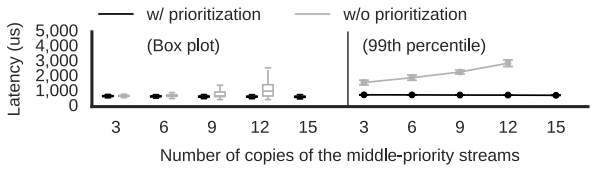
Platform: Our test-bed consists of three machines: one running all event suppliers (Pentium Dual-Core 3.2 GHz, Ubuntu Linux with kernel v.3.19.0), one running the CPEP event service (Intel i5-4590 3.3 GHz four-core machine, Ubuntu Linux with kernel v.4.2.0), and one running all event consumers (Pentium Dual-Core 3.2 GHz, Ubuntu Linux with kernel v.3.13.0). We connected the three machines via a Gigabit switch running in a closed LAN. The machine running CPEP had two NICs, and we used one for inbound traffic and another for outbound traffic. Out of its four cores, three cores were dedicated to event processing and one core was dedicated to proxy inbound traffic. 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 that proxies the inbound traffic. We did not use the CONFIG_PREEMPT_RT patch [12].

Workload and Consumer Configuration: We explored the effectiveness of CPEP by feeding events at different rates and processing streams of different priorities. We used two graphs of processing streams, and in each setting we conducted sub-cases to show the performance under different degrees of system workload. We simulated the operation of Multi-sensor fusion, using the following four operators: Extended Kalman Filter (EKF) [13], Fast Fourier Transform (FFT) [14], Concatenation (CAT) (implemented using C++'s `memcpy` function), and the Advanced Encryption Standard (AES) [15]. We coordinated the event suppliers in the same stream to reduce unnecessary latency due to time differences among the pushes from distinct suppliers.

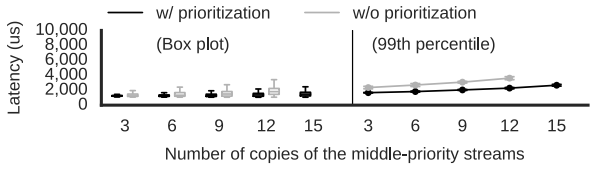
Method of Measurement: We ran each sub-case ten times and calculated the 95% confidence interval for each measurement. In each sub-case we sequentially ran three phases: warm-up, measuring, and dumping. The warm-up phase took ten seconds, during which we connected all event suppliers and consumers to CPEP and had them start pushing and receiving events; the measuring phase took 100 seconds, during which we measured both the latency and the throughput of output events, and we kept all the measurements in memory, which were then saved to disk in the dumping phase. We measured the end-to-end latency, i.e., 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 Fig. 2).

A. Experiment 1: Prioritization

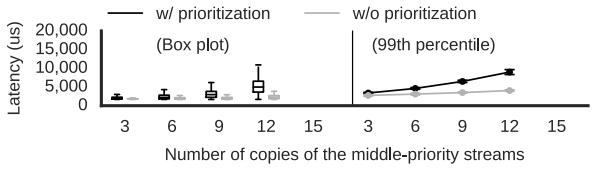
In this experiment we compared the event latency of prioritized processing versus that of non-prioritized processing. Fig. 5 shows the graph of event processing streams. To cover different degrees of workload, we first deployed three copies of the whole graph and then increased the workload by deploying more copies of the middle-priority streams. Each supplier event carried a batch of one-byte datapoints: each event supplied by s_1 and s_2 carried 512 datapoints (200 Hz event rate); s_3 to s_6, 1024 datapoints (100 Hz); and s_7 and s_8, 2048 datapoints (50 Hz). Each output event for a high-



(a) High priority.



(b) Middle priority.



(c) Low priority.

Fig. 6. Experiment 1: Latency under different loads; y-axis limit is set in accord to the event rate; data exceeding the limit was not shown.

priority consumer carried 512 one-byte datapoints; each output event for a middle-priority consumer carried 1024 16-byte datapoints[2]; each output event for a low-priority consumer carried 2048 eight-byte datapoints.

Fig. 6 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 Fig. 6(a), 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 Fig. 6(b). Low-priority streams exhibited the opposite behavior, shown in Fig. 6(c), where prioritization led to higher latency 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 period and, as the next experiment will show, sharing operations can further reduce the latency.

Fig. 6 also shows that streams may have high tail latency even though the system was not heavily loaded (for example, the sub-case of six middle-priority streams, with the normalized CPU utilization 63%). This happened because event arrivals of different streams were independent of each other and sometimes arrived close in time and contended 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.

[2]The FFT operator caused the increase in datapoint size, as it transformed each byte of datapoints into an eight-byte real number (we used the single precision version of FFTW).

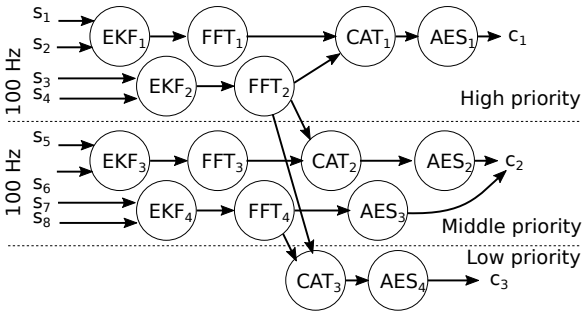
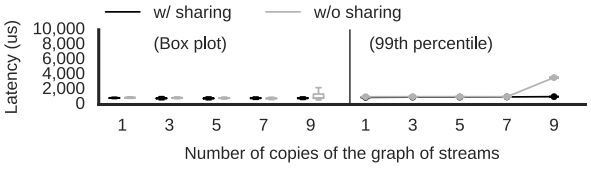
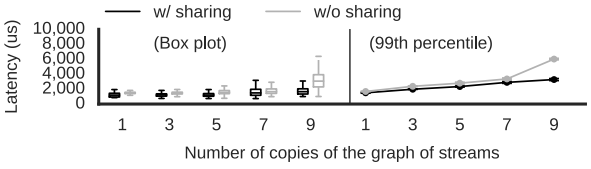


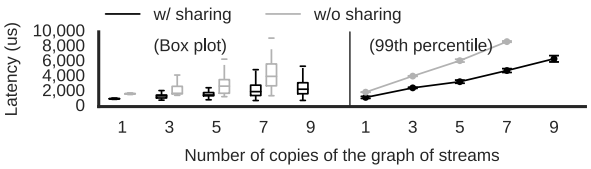
Fig. 7. Experiment 2: The graph of event processing streams.



(a) High priority.



(b) Middle priority.



(c) Low priority.

Fig. 8. Experiment 2: Latency under different loads; y-axis limit is set in accord to the event rate; data exceeding the limit was not shown.

The sub-case of 15 middle-priority streams shows that with prioritization, higher-priority streams may keep producing events with low latency; without prioritization, however, no stream could produce events with the latency lower than the sending period of suppliers.

B. Experiment 2: Sharing and Shedding

In this experiment we first evaluated the latency performance of sharing operations, then we stress-tested the system and evaluated the shedding strategy and sharing operations in terms of *timely-throughput*, i.e., delivery of events within their timing constraints. In both cases we also enabled prioritization. We configured the graph as shown in Fig. 7. The non-sharing version (for comparison) was constructed by duplicating the shared operators (FFT_2 and FFT_4) and all their upstream operators (EKF_2 and EKF_4). All suppliers generated events at a rate of 100 Hz, with the absolute validity interval set to 10 ms. Because the streams share operations, here we varied workload by deploying copies of the whole graph.

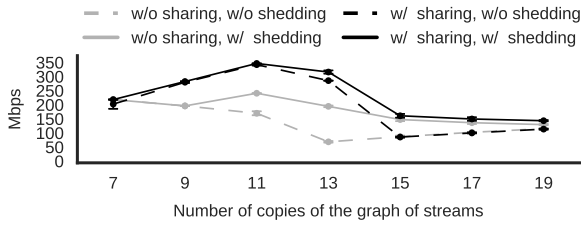


Fig. 9. Experiment 2: Total timely-throughput (Mbps).

The latency results shown in Fig. 8 confirm that sharing operations can help reduce event latency. Lower-priority streams received higher reductions, because the shared operations were done by the higher-priority counterpart. The result also suggests that the savings in processing time may outweigh the time spent waiting for higher-priority streams, and sharing only introduced negligible overhead to higher-priority streams.

Fig. 9 shows the timely-throughput in terms of Megabits per second (Mbps) 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 200 Mbps with up to 19 copies of the graph of streams; with sharing, the normalized utilization approached 80% in the presence of 11 copies of the graph, and then as we increased the workload, the benefit of shedding prevailed, giving 100 additional Mbps in the presence of 15 copies of the graph of streams. The benefit of shedding gradually diminished as we deployed more copies, because the system was being dominated by only high priority streams.

VI. Related Work

Cyber-physical event processing is an essential part of modern Industrial Internet-of-things architectures [16], and in many use cases [3], [4] it is critical to minimize the time it takes to respond to stimuli. To this end, both messaging middleware (for example, Kafka [17]) and the Data Distribution Service (DDS [18]) 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' QoS. DDS provides QoS options for data delivery, but does not process data. In contrast, CPEP both differentiates messages according to consumers' priority levels and processes data subject to absolute time consistency.

The field of Complex Event Processing (CEP) [19] offers a rich set of semantics for expressing stimuli using sets of events [20], [21]. GraphCEP [22] is such a system and processes events for social network analysis. The system 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 and does not share computation. In contrast, CPEP maintains time consistency at the timescale of microseconds and supports sharing of computation between processing streams.

VII. Concluding Remarks

In this paper, 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 on multi-core machines, showing that CPEP can both reduce processing latency and improve temporally valid throughput.

Acknowledgment

This research was supported in part by NSF grant 1329861 and ONR grant N000141612108.

References

[1] "Connext DDS at a glance: understanding the software framework that connects the Industrial IoT," White Paper, Real-Time Innovations, 2017.
[2] P. C. Evans and M. Annunziata, "Industrial internet: pushing the boundaries of minds and machines," *General Electric Reports*, 2012.
[3] D. Kirsch, "The value of bringing analytics to the edge," *Hurwitz & Associates*, 2015.
[4] (2017) Industries served and use cases - Foghorn Systems. [Online]. Available: https://foghorn-systems.com/industries/
[5] N.-E. El Faouzi, H. Leung, and A. Kurian, "Data fusion in intelligent transportation systems: progress and challenges–a survey," *Information Fusion*, vol. 12, no. 1, pp. 4–10, 2011.
[6] A. Sebastian and A. Pantazi, "Nanopositioning with multiple sensors: a case study in data storage," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 2, pp. 382–394, 2012.
[7] D. Piri, "Sensor fusion for nanopositioning," Master's thesis, Vienna University of Technology, Austria, 2014.
[8] (2017) The ADAPTIVE communication environment. [Online]. Available: http://www.cs.wustl.edu/%7Eschmidt/ACE.html
[9] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The design and performance of a real-time CORBA event service," *ACM SIGPLAN Notices*, vol. 32, no. 10, pp. 184–200, 1997.
[10] J. A. Stankovic, S. H. Son, and J. Hansson, "Misconceptions about real-time databases," *Computer*, vol. 32, no. 6, pp. 29–36, 1999.
[11] G. R. Lavender and D. C. Schmidt, "Active object: an object behavioral pattern for concurrent programming," in *Proc. Pattern Languages of Programs,*, 1995.
[12] (2017) The CONFIG_PREEMPT_RT patch. [Online]. Available: https://rt.wiki.kernel.org
[13] (2017) The KFilter Project. [Online]. Available: http://kalman.sourceforge.net
[14] (2017) FFTW. [Online]. Available: http://www.fftw.org
[15] (2017) The Libgcrypt Library. [Online]. Available: https://gnupg.org/software/libgcrypt
[16] *Industrious Internet Reference Architecture*, Industrial Internet Consortium Std., Rev. 1.8, Jan 2017.
[17] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: a distributed messaging system for log processing," in *Proceedings of the NetDB*, 2011, pp. 1–7.
[18] (2017) Data distribution service (DDS). [Online]. Available: http://www.omg.org/spec/DDS/
[19] 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.
[20] 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*. ACM, 2010, pp. 26–37.
[21] 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.
[22] R. Mayer, C. Mayer, M. A. Tariq, and K. Rothermel, "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, 2016, pp. 309–316.