Reliable Speculative Processing of Out-of-Order Event Streams in Generic Publish/Subscribe Middlewares

Christopher Mutschler[1,2]
christopher.mutschler@fau.de

Michael Philippsen[1]
michael.philippsen@fau.de

[1] University of Erlangen-Nuremberg
Department of Computer Science
Programming Systems Group
Erlangen, Germany

[2] Fraunhofer Institute for Integrated Circuits IIS
Locating and Communication Department
Sensor Fusion and Event Processing Group
Erlangen, Germany

ABSTRACT

In surveillance, sports, finances, etc., distributed event-based systems are used to detect meaningful events with low latency in high data rate event streams. Both known approaches to deal with the predominant out-of-order event arrival at the distributed detectors have their shortcomings: buffering approaches introduce latencies for event ordering and stream revision approaches may result in system overloads due to unbounded retraction cascades.

This paper presents a speculative processing technique for out-of-order event streams that enhances typical buffering approaches. In contrast to other stream revision approaches our novel technique encapsulates the event detector, uses the buffering technique to delay events but also speculatively processes a portion of it, and adapts the degree of speculation at runtime to fit the available system resources so that detection latency becomes minimal.

Our technique outperforms known approaches on both synthetical data and real sensor data from a Realtime Locating System (RTLS) with several thousands of out-of-order sensor events per second. Speculative buffering exploits system resources and reduces latency by 40% on average.

Categories and Subject Descriptors

C.2.4 [**Computer-Comm. Networks**]: Distrib. Syst.—*Distrib. Applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming*

Keywords

Distributed Event Processing; Out-of-Order Event Processing; Publish/Subscribe; Message-oriented Middleware.

1. INTRODUCTION

Event-based systems (EBS) are the method of choice for a near-realtime, reactive analysis of data streams in many fields of application such as surveillance, sports, stock tradings, RFID-systems, and fraud detection in various areas [1]. EBS turn the high data load into events, and filter, aggregate, and transform them to higher level events until they reach a level of granularity that is appropriate for an end user application or to trigger some action. Often the performance requirements are high so that event processing needs to be distributed over several nodes. Many applications also demand event detection with minimal delays. For instance, our distributed EBS detects events to steer an autonomous camera control systems to points of interest, see Fig. 1. This obviously requires low detection latencies.

To process high rate event streams, an EBS usually splits the computation over several event detectors, linked by publish/subscribe to build an event detection hierarchy. These event detectors are distributed over the available machines. Events arrive at the distributed detectors out-of-order because of various types of delay. Ignoring the wrong order causes misdetection. Event detectors themselves cannot reorder the events with low latency because in general event delays are unknown before runtime. Moreover, as there are also dynamically changing application-specific delay types (like for instance a detection delay) there is no a-priori optimal assignment of event detectors to available nodes. Hence, in a distributed EBS the middleware deals with out-of-order events, typically without any a-priori knowledge on the event detectors, their distribution, and their subscribed events.

Buffering middleware approaches withhold the events for some time, sort them, and emit them to the detector in order. The main issue is the size of the ordering buffer. If it is too small, detection fails. If it is too large, it wastes time and causes high detection latency. Note that waiting times add up along the detection hierarchy. The best buffer sizes are unknown and may depend on some dynamic, unpredictable behavior. In addition, there is no need to buffer

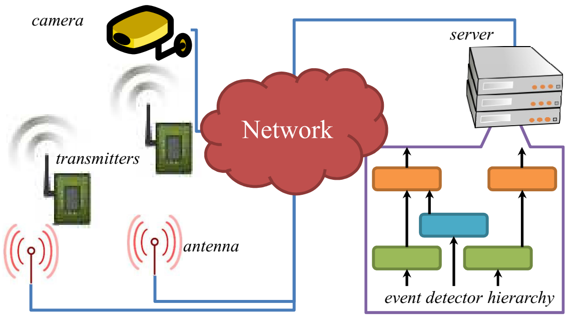


Figure 1: Automatically controlled camera system.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DEBS'13, June 29–July 3, 2013, Arlington, Texas, USA.
Copyright 2013 ACM 978-1-4503-1758-0/13/06 ...$15.00.

events that cannot be out-of-order or that can be processed out-of-order without any problems. Buffering middlewares are the basis of reliable event detection, but they are also too costly for many types of events and do not benefit from faster CPUs (as they are bound by the waiting times).

Speculative middlewares, the other approach to cope with out-of-order event arrivals, speculatively work on the raw event stream. As there is no buffering, this is faster. Whenever an out-of-order event is received, falsely emitted events are retracted and the event stream is replayed. The effort for event retraction and stream replay grows with the number of out-of-order events and with the depth of the detection hierarchy. This is a non-trivial challenge for the memory management, may exhaust the CPU, and may cause high detection latencies or even system failures. In contrast to buffer-based approaches, a stronger CPU helps, but the risk of high detection latencies remains.

To combine the advantages of both, this paper proposes to add a novel speculative processing to a buffering EBS. The most important requirements are: (1) The middleware can neither exploit the events' semantics nor their use by the event detectors because both of them are highly application-specific. Hence, methods for strong synchronization of event detectors are no viable option. (2) In spite of the speculation, the buffering middleware must keep event detection reliable, i.e., false-positive or false-negative detection must be avoided to prevent system failures. Hence, it is no option to use imprecise approximative methods or to discard events that would cause a system overload.

Our key idea is to use buffering to sort most of the events but to let a snapshot event detector speculatively and prematurely process those events that will be emitted soon. Event detectors are restored when a replay occurs. The degree of speculation is adapted to suit the CPU availability, ranging from full speculation on an idle CPU to plain buffering on a busy CPU. Our technique works without knowledge on internal event semantics, can be used for any publish/subscribe-based buffering middleware, and does not use query languages or event approximation.

The rest of the paper is organized as follows. Sec. 2 reviews related work. Sec. 3 provides basic definitions of the time-model and introduces the K-slack buffering that we use in the rest of the paper to present the novel speculation. The main contributions of this paper are covered in Sec. 4: a speculative event processing and snapshot recovery, two methods to efficiently retract events across the detection hierarchy, and a greedy method to adapt the amount of speculation at runtime to optimize detection latency by efficiently exploiting unused system resources. Sec. 5 evaluates our methods before Sec. 6 concludes.

2. RELATED WORK

As we know of no attempt to fully combine buffering and speculation in an EBS, we discuss both fields in turn.

2.1 Buffering Techniques

As we have explained above, buffering is needed for reliable event detection but also introduces latency. Below we sketch some known buffering EBS and show that for each of them an added speculative component might improve latency.

To deal with communication errors and timing uncertainties caused by the lack of a global clock in distributed systems O'Keeffe et al. [2] use time intervals instead of event time stamps. Buffered events are only emitted when time intervals safely overlap. Therefore, as latencies are high, an added speculative component that lowers the upper interval margin can reduce detection latency.

Punctuations [3, 4] are special annotations embedded into data streams to indicate (1) the end of a subset of data, and (2) that no event will be generated with a lower time stamp. For negative patterns like `A!BC`, where no `B` shall occur in between `A` and `C`, a buffering unit must use timed punctuations that introduce latency because firing a punctuation on each event will exhaust the system. An added speculation unit can help because punctuations can also be fired speculatively, so that events can be consumed earlier.

Srivastava et al. [5] model stream time differences and clock-skews between data sources. Their heartbeat stream synchronization among buffers introduces latency. Speculation can help by processing events before the heartbeats actually occur or by emitting the heartbeats prematurely.

SASE [6, 7, 8] and Cayuga [9, 10, 11, 12] both are event processing engines for RFID-readings that work with Nondeterministic Finite Automata generated from event queries. They assume that events are delayed for at most K time units, K to be set a-priori. Although both systems use query languages they can be applied for arbitrary middlewares. But their conservative and a-priori configuration causes high detection latencies in their delay buffers. Speculation can help by processing events before they have been buffered sufficiently long, i.e., for K time units.

2.2 Speculative Techniques

Known EBS speculation techniques are either based on query languages and window operators, are imprecise and approximate events, are unreliable, or use a-priori knowledge for their configurations. Hence, as they do not fulfill all the above-mentioned requirements for general purpose publish/subscribe middleware, known speculation techniques cannot be used as add-ons to a buffering system.

Approximative techniques [13, 14, 15, 16] use partial or imprecise events to generate the most likely query results first and refine the results incrementally on corrected and more precise events. To limit latency and resource consumption, some of the authors only retract events that have a strong impact on the generated output or use partial events to restore the correct state for the replay. However, as all those systems use query languages and are imprecise, they are unsuitable as a generic speculation component.

Another way to limit the degree of speculation (and hence latency and resource consumption) is to automatically discard older and unprocessed events from their queues and to process more recently received events instead [17]. However, this is only applicable to state-less event detectors. In general, event detector states depend on the discarded events so that detection fails if they are dropped.

A commit action resolves the speculation (the essence of transaction systems) into a reliable event detection step [18, 19]. But every out-of-order event increases the number of transactions, i.e., the degree of speculation, and results in potential CPU overload and hence system failure. That is why buffering EBS stay away from transaction speculation.

Window query approaches [20] pipe events through operator graphs that are constructed from queries. The degree of speculation is set by limiting historical operator states.

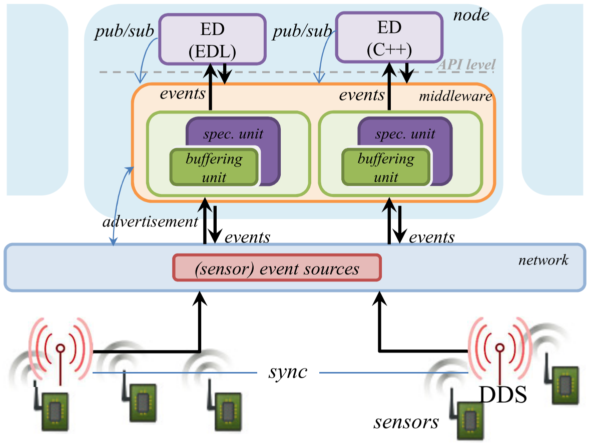


Figure 2: Distributed publish/subscribe EBS.

Unfortunately, window operators (such as multi-joins) need event detector definitions in a particular query language and are hence not general purpose.

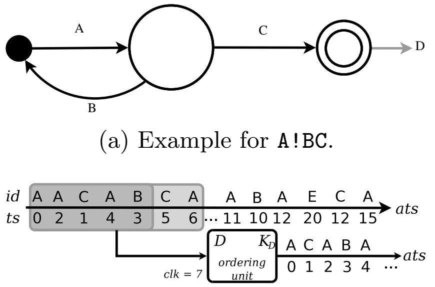
Complex Event Detection and Response (CEDR) [21] speculatively generates events out of event streams that also may contain revision tuples enhanced with time stamp corrections. Upon a revision event arrival, CEDR only adjusts the time stamps of the generated event instead of retracting the event. But this brakes stateful event detectors that may already have consumed the generated event with the old time stamp, and may have generated a (false-positive) event because of the old time stamp.

Chandramouli et al. [22] limit speculation either by sequence numbers or by *cleanse*. The receiver can use the former to deduce disorder information in the rare cases when particular events are generated at stable rates. The latter only works for a punctuation-based environment, which must incorporate the event definition to limit query windows by setting the punctuation to the latest event time stamps of the event detector. Since the middleware technique cannot access this information, it cannot be used as a generic buffering extension.

3. BUFFERING EBS ENVIRONMENT

In this section, we illustrate the buffering in the hosting EBS that we extend with our novel speculative technique. Although Sec. 4 will build on top of this buffering, the novel speculation can easily be adopted by other buffering EBS because our speculation is general purpose and quite generic.

As can be seen in Fig. 2, our EBS consists of several data distribution services (DDS) that collect sensor data (for example an antenna that collects RFID readings), and several nodes in a network that run the same event processing middleware. The middleware creates a reordering buffer per event detector (ED). This is wrapped with the new speculation unit. The middleware deals with all types of delays such as processing and networking delays or detection delays[1] and does not need to know the complex event pattern that the detector implements (either in a native programming language [23] or in some EDL [6]). The detector does not need to know on which machine other event detectors are running nor their runtime configurations. The application code of the event detector assumes that the events are received in correct order with respect to their occurrence time stamps. At startup the middleware has no knowledge about event delays but just notifies other middleware instances about event publications and subscriptions (advertisement) [24]. The middleware is therefore generic and encapsulated.



(a) Example for `A!BC`.

(b) Sorting window over event stream.

Figure 3: Out-of-order examples.

Since our speculative buffer asks the event detector to provide and to restore snapshots, event detectors have to provide additional functionality. However, in many cases this can easily be achieved as snapshots are handled transparently to the (application) code in the event detector that is used to process the event stream.[2]

3.1 Time Model Semantics

We use the following terminology throughout the paper:
Event type, instance and time stamps. An event type defines an interesting occurrence and is identified by a unique ID. An event instance is an instantaneous occurrence of an event type at a point in time. It can be a primitive (sensor) or a composite event. An event has three time stamps: an occurrence, a detection, and an arrival. All time stamps are in the same discrete time domain according to our time model. An event appears at its occurrence time stamp ts, or just time stamp for short. It is detected at its detection time stamp dts. At arrival time stamp ats the event is received by a particular EBS node. The occurrence and the detection time stamp are fixed for an event at any receiving node whereas the arrival time stamp may vary at different nodes in the network.
Out-of-order event. Consider an event stream $e_1, \cdots e_n$. Events of type `ID` are used to set the local clock. Then e_j is out-of-order if there do not exist e_i, e_k, with $e_i.id{=}e_k.id{=}$`ID` and $e_i.ats{\leq}e_j.ats$ so that $e_i.ts{\leq}e_j.ts{\leq}e_k.ts$, i.e., e_j.ats does not fit between two consecutive clock updates, see Sec. 3.2.
Event Stream. The input of an event detector is a potentially infinite event stream that usually is a subset of all events, holds at least the event types of interest for that detector, and may include some irrelevant events as well.

3.2 K-slack in a detector hierarchy

Consider the following example. To detect that a player kicked a ball, we wait for the events that a ball is near the player and then, that the ball is kicked, a peak in acceler-

[1] If events are generated with earlier time stamps than the time stamps of the events that cause them, they have a detection delay and can only be inserted into the event stream long after they have actually happened.

[2] The application code does not need to care about data synchronization or side-effects as the middleware assures that no events are being processed by the detector while taking or restoring a snapshot.

RIGHTSLINK

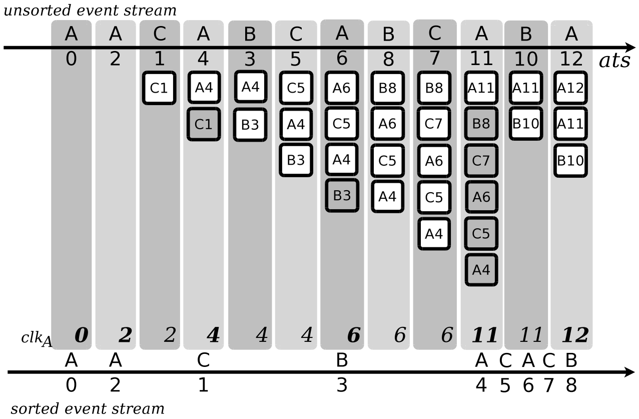


Figure 4: Ordering unit for event detector A!BC

ation. Between the two events there may not be the event that the ball leaves the player, because in that case the ball would just have dropped to the ground. More formally: if we receive event A (near) and subsequently C (acceleration peak) and not B (not near) in between, we generate event D.[3] Figure 3(a) gives a finite state automaton for event D. To simplify, we leave out the differentiation of transmitter IDs for player identification.

An exemplified event stream is depicted in Figure 3(b). The events in the stream are out-of-order and a naive processing of events will not lead the event detector to generate event D out of A4/C5 and will detect D out of A2/C1.[4]

To achieve a correct detection, the buffering middleware of our hosting EBS mounts a dynamically generated ordering unit based on K-slack between the event input stream and the event detector. K-slack [7, 25, 26] assumes that an event e_i can be delayed for at most K time units.

K-slack is a standard approach that works as follows (with dynamic buffer-resizing). A local clock clk is extracted out of the event stream by setting it to the occurrence time stamps of particular event types [26]. Hence, whenever clk is updated we can deduce the maximal delay of all events e_i that have been received since the last clk update by $\delta(e_i)$=clk-$e_i.ts$. Hence, for a particular event detector the ordering unit (that takes a stream with potential out-of-order events and produces a sorted event stream) must set K to the maximal delay of all its subscribed events, i.e., K=max$_i$ $[\delta(e_i)]$. This gives the initial size of the dynamically-sized buffer for event ordering. Hence, whenever we receive a new event we just insertion-sort it into the buffer. Whenever clk is updated and $e_i.ts+K\leq clk$ holds for some tail events e_i in the buffer, we emit those e_i to the output stream and purge them from the buffer as we can be sure that there will not be any event to be received in the future that has a time stamp lower than $e_i.ts$.

Fig. 4 shows the internals of the event ordering unit for the input stream of Figure 3(b). clk is set whenever an event of type A is received, see the bold values in the clk_A line. At the beginning when A0 and A2 are received, there are no measurements and K is still 0, which means that both events are immediately passed to the output stream, without being delayed at all (they fulfill $e_i.ts+K\leq clk_A$). When C1 is received, it is pushed to the buffer and waits until A4 updates clk_A. As the delay for C1 is 3=4-1, we set K=3 and emit C1 ($e_i.ts+K$=1+3$\leq clk_A$ =4). A4 is buffered at least until clk_A equals 7=4+K. With A6 the maximal delay of B3 is 3=6-3, K holds, and B3 is passed to the output stream. Hence, we extract both a local clock clk_A and K from the event stream to delay both late and early events as long as necessary to avoid out-of-order event processing at the event detector (after an initial calibration of the ordering unit).

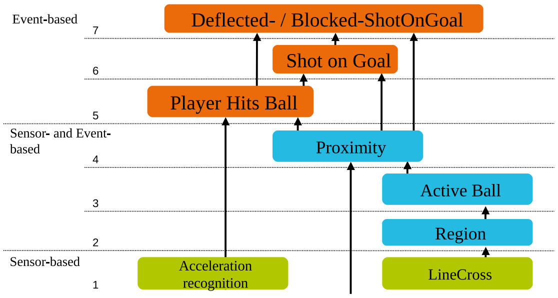


Figure 5: Event processing hierarchy.

Fig. 5 shows an event processing hierarchy for a *blocked shot on goal.* Each event detector in this hierarchy has its own dynamically parameterized ordering unit, and is configured to detect events with low latency. For instance, the *player hits ball* (level 5) event detector implements a pattern similar to that of Fig. 3(b). Its ordering unit introduces a latency of more than one second to guarantee an ordered event input to the detector. This delays the detection of a *shot on goal* (level 6) for at least one second (and the *blocked shot on goal* may be delayed even longer).

3.3 Motivation

Even if we use minimal K-values for all the detectors across the hierarchy, the resulting combined latencies may be unnecessarily high and a speculative processing may be the better option.

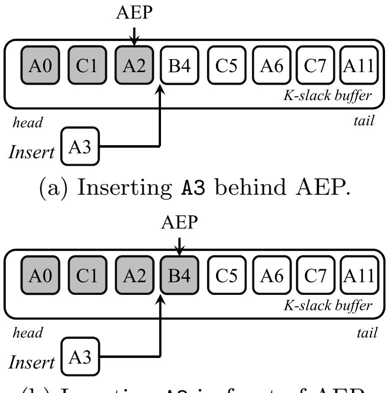
Assume the example in Fig. 3(b) needs a minimal K_D=3. This delays the detection of any event in upper processing hierarchies by at least 3 time units since event D can only be detected with 3 time units delay. However, assume that events of type B are rare. Then it may be advantageous not to delay the detection of D until we preclude the occurrence of B, but to retract a false-detection of D in the rare cases when B actually occurs. For the event stream in Fig. 3(b) we can hence detect D out of A4/C5 before clk is set to 8 (A4 is emitted at clk=7 whereas C5 must wait at least until $clk\geq$8). If there is a B to cancel the detection of D later we retract D. Hence, the event detector that is used to detect D generates preliminary events that can be used to trigger event detectors on higher levels with low latency.

Hence, the key idea is to combine both techniques and to let the speculation unit wrap a K-slack buffer to process a portion of events prematurely.

4. SPECULATIVE PROCESSING

Added speculation results in improved detection latency if there are no out-of-order events at all, because nothing that an ordering unit emits and a detector generates has ever to

[3] This is similar to the book shelf reading example that is often used in applications combining RFID systems and EBS.
[4] Certainly this problem only arises when events are merged. However, this is necessary because we split computing across several event detectors and must iteratively summarize preliminary results (events).



(a) Inserting `A3` behind AEP.

(b) Inserting `A3` in front of AEP.

Figure 6: K-slack's event insertion and AEP.

be retracted from further up the detector hierarchy. But the more out-of-order events there are and the deeper the detection hierarchy is, the more complex becomes the retraction work as more memory is needed to store the detector states and as more CPU time is needed to perform the retraction. Hence, in naive speculation approaches, the cost of purging the effects of false speculation can easily outweigh its beneficial effects and can easily increase the latency beyond what pure non-speculative buffering would have caused.

Therefore, the amount of speculation must be limited so that the CPU and memory are used at full capacity on the one side, but without getting exhausted on the other side. System parameters must be deduced at runtime and the speculative ordering units must be continuously adapted to the current system and event load. From now, we use the following terminology:

Event emission and replay. The ordering unit emits events to the event detector to process them. An event is emitted *prematurely* if it is emitted before K-slack would emit it. When an event detector produces an event and sends it to the middleware we say an event is *generated*. Whenever the ordering unit detects a miss-speculation the event stream is replayed. In those cases particular (premature) events are emitted repeatedly to the event detector, see Sec. 4.1.

Event retraction. Whenever the ordering unit detects a miss-speculation, events may have been mistakenly processed and need to be retracted. We have to consider two different issues. First, events have been emitted to the event detector in a wrong order and the event stream is immediately replayed. This is solved by *snapshot recovery*, see Sec. 4.2. Second, the event detector may have generated events based on the incorrectly ordered event input. These events may already have been inserted into the ordering units' buffers of upper level event detectors or may even have been emitted prematurely. Hence, events must be retracted throughout the entire detector hierarchy, see Sec. 4.3.

4.1 Event Emission and Replay

Our speculative event processing technique extends K-slack buffering approches. It puts most of the input events in order but it does not buffer them as required for a perfectly correct order. Instead of buffering an event e_i for K time units, we only buffer e_i as long as

$$e_i.ts + \alpha \cdot K \leq clk, \qquad \alpha \in [0;1], \qquad (1)$$

with a new attenuation factor α. The attenuation factor

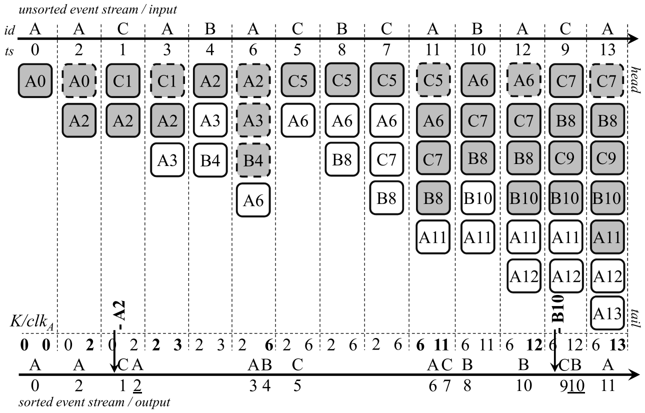

unsorted event stream / input
id
ts
A A C A B A C B C A B A C A
0 2 1 3 4 6 5 8 7 11 10 12 9 13
head
tail
K/clk_A
- A2
- B10
sorted event stream / output

Figure 7: Speculative ordering unit with $\alpha = \frac{1}{3}$.

is used to adjust the speculation component. The larger α is, the fewer events are emitted prematurely, i.e., α=1 is essentially a K-slack without speculation. Smaller values for α switch on speculation. For α=0, there is no buffering/ordering at all because the inequation always holds (except for events with negative delays[5]). For instance, a classic event ordering unit with K=5 and α=1 will emit an event with ts=20 that is received at clk=22 not before clk is at least 25. Only then e_i.ts+K=20+5≤25=clk. Pure buffering middlewares will not just emit the events but they will also purge them from the buffer. In the example with K=5 but with α=0.6 the speculative buffer prematurely emits the event already at clk=23 (20+0.6·5=23). With α≤0.4 emission is even instantly at clk=22.

With speculation, events are emitted but no longer instantly purged from the buffer. They may be needed for the event replay later. Hence, the K-slack buffer is enhanced with an *already emitted pointer* (AEP) that is a reference to the last element in the buffer that has already been emitted speculatively.

A newly arriving event is inserted into the sorting buffer according to its time stamp. For instance, in Fig. 6(a) and 6(b) `A3` is inserted between `A2` and `B4`. The events from the buffer's head to AEP (shown in grey) have already been emitted. Depending on α, clk, K, and AEP there are two possible cases:

$\mathbf{e_i.ts > e_{AEP}.ts}$: If the time stamp of the recent event e_i is larger than that of the AEP, no false-speculation has occurred, and hence no events need to be retracted or replayed. In Fig. 6(a) `A0` to `A2` have been prematurely emitted, and `A3` is not missing in the stream that is already on the go.

$\mathbf{e_i.ts \leq e_{AEP}.ts}$: If the time stamp of the new event is smaller that that of the AEP the falsely emitted events must be retracted by means of the methods that we describe in Sec. 4.3. The AEP is set to e_i and the events from the buffer's head to the new AEP are replayed. In Fig. 6(b) the event `B4` must be retracted, and `A3` can be emitted prematurely before replaying `B4`.

Whenever clk is updated the speculative buffer emits all the events that fulfill inequation (1). Events are purged only if the regular non-speculative K-slack buffer would purge them.

Fig. 7 shows how the speculative buffer for the event detector from Fig. 3(a) works with α=1/3 and an initial K=0.

[5] This may be the case if different events are used to set clk.

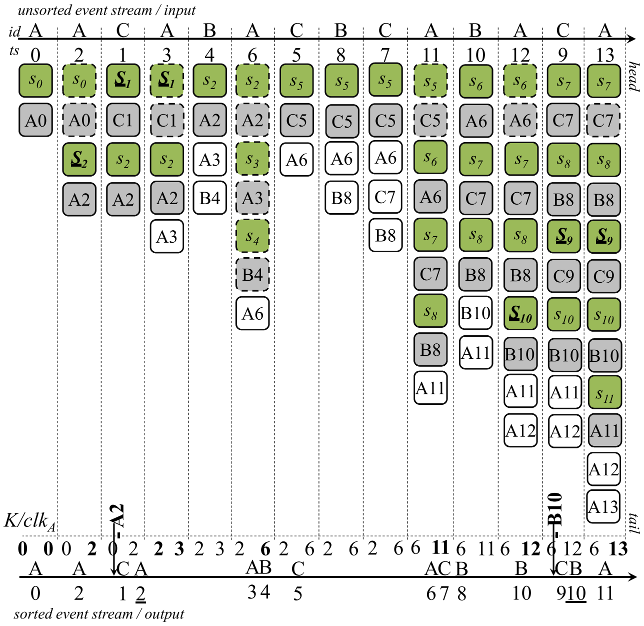


Figure 8: Speculative ordering unit with snapshot recovery and $\alpha = \frac{1}{3}$.

The orientation of the buffer is now vertical, the head is on top. Events of type A are used to set the internal clock clk_A, and K is adjusted by means of dynamic K-slack, see the bold values. Dashed events will be purged from the buffer. Prematurely emitted events are in grey. The AEP, although not explicitly visualized, points to the lowest grey event. Retraction is shown with negative events that point to the output stream. Replayed events are underlined.

At the beginning, we have no measurements of event delays, so we emit A0 and A2 (speculatively). When C1 is received, we detect that we incorrectly emitted A2 before, and replay the correctly ordered sub-stream, i.e., we emit C1, and again A2. K is set to 2 as soon as A3 arrives. A3 is not yet emitted, because $e_i.ts+\alpha\cdot K=3+\frac{1}{3}\cdot 2=3.67>3=clk$. C1 can now be purged as K=2 tells us that there cannot be out-of-order events with $ts<3\text{-}K=1$. We insert B4 into the buffer. With A6, we evaluate the currently buffered events, emit A3 ($3.67\leq 6=clk_A$) and B4 ($4.67\leq 6=clk_A$), and purge the buffer by erasing each event that satisfies $e_i.ts+K\leq clk$, i.e., A2, A3, and B4 that are now safe (by means of the rules of pure K-slack). Although clk is not updated on reception of C5, we can prematurely emit C5 because it fulfills $e_i.ts+\alpha\cdot K=5+\frac{1}{3}\cdot 2=5.67\leq 6=clk$, and since A6 has not been processed before, we do not need to replay. B8 and C7 are both queued until reception of A11. We then process A6, C7, and B8, and purge C5. With A12 we prematurely emit B10 because $e_i.ts+\alpha\cdot K=10+\frac{1}{3}\cdot 6=12\leq 12=clk$. We insert C9 in front of the AEP and thus detect another false speculation. Hence, we relocate the AEP and replay C9 and B10.

Whenever the event detector generates an event out of a prematurely emitted event, the middleware attaches the original K-value difference, i.e., K^+, so that the ordering units of higher level event detectors can calculate the event delay, and hence their new K-value, appropriately.

4.2 State Recovery

If speculation was too hasty the speculative ordering unit relocates the AEP and replays the event stream. However, although the ordering unit may revise incorrect events by means of the retraction methods that we describe in Sec. 4.3, the event detector that processes the emitted events may still be in a wrong state due to these events. Hence, for a replay the internal variables of the event detector must be reverted to the state they had before the event detector processed the first incorrect premature event to avoid inconsistencies.

Algorithm 1: Adding a newly received event e.

```
Data: Event $e$, OrderingBuffer buffer, Mutex $m$,
      WorkerThread workerThread
begin
    UpdateK($clk$-$e.ts$+$e_i.K^+$); // update $K$ if needed
    while !m.acquireLock() do                  // lock buffer
        workerThread.interrupt();
    BufferIterator it ← buffer.tail;
    repeat
        it ← it.previous;
        if it.GetTime() $\leq$ e.GetTime() then
            buffer.insertAfter(it, e);
            break;
    until it = buffer.head;
    if e.GetTime() < buffer.head.GetTime() then
        buffer.push_front(buffer.head, e);
    if AEP.ts > e.ts then      // it.next is snapshot
        Event s ← it.next.pop();
        s.SetTime(e.GetTime());               // adjust ts
        buffer.insert_before(it, s);       // move snapshot
        buffer.emit(s);                 // re-init detector
        buffer.SetAEP(it);                  // relocate AEP
    m.releaseLock();
    workerThread.wakeUp();
```

Such a state recovery is difficult because of three reasons. First, since the ordering middleware transparently handles out-of-order event streams the event detector does not even know that an ordering unit exists. Second, even if the event detector knows that there is a speculative ordering unit, and it processes retraction events to revert its state, it nevertheless has no clue about α and K, and hence how many states it needs to keep. Third, in many cases retraction cascades, which are the core reason why speculation must be limited, can be interrupted earlier and resolved faster. But this is only possible from within the ordering middleware, see Sec. 4.3.2.

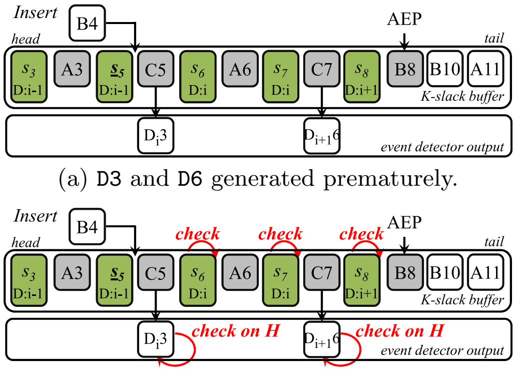
Our solution is to let the middleware trigger both state backup and recovery. This avoids side-effects by out-of-order events in the detector and the application programmer does not need to care about retraction events or recovery. On demand, the event detector has to be able to provide all the data that is necessary to later on be restored to this snapshot. The key idea is to ask the event detector for a snapshot whenever a premature event e_i is going to be emitted and to insert this snapshot as an exceptional event e_s with $e_s.ts$=$e_i.ts$ into the ordering buffer, directly in front of the prematurely emitted event e_i. The snapshot then represents the event detector state that has been in place before e_i was prematurely emitted.

Whenever events are replayed to an event detector, the detector switches back to an earlier state as soon as it receives such an exceptional event e_s encapsulating that earlier state. Only the first/earliest buffered snapshot event is emitted in

RIGHTSLINK

Algorithm 2: Event emission, replay, and buffer purge.

```
Data: OrderingBuffer buffer, Mutex m, Clock clk
begin
    while true do
        if m.acquireLock() then
            while $AEP \neq$ buffer.tail() do
                CheckAndBreakOnInterrupt();
                if $AEP.GetTime() \leq clk - \alpha \cdot K$ then
                    SnapshotEvent $s \leftarrow$ MakeSnapshot();
                    buffer.insert_before(AEP, s);
                    buffer.emit(AEP);
                    $AEP \leftarrow AEP$.next();
                    if AEP.isSnapshotEvent() then
                        $AEP \leftarrow AEP.next$;
                else
                    break;
            // buffer purge by K-slack constraints
            while $buffer.head().GetTime + K < clk$ do
                if $buffer.head = null$ then
                    break;
                if $buffer.head() \neq AEP$ then
                    buffer.pop_front();
                else
                    break;
            m.releaseLock();
        else
            m.sleep();
```



(a) D3 and D6 generated prematurely.

(b) On-Demand Retraction.

Figure 9: Event retraction on reception of B4.

a replay, the remaining snapshots are skipped. During a replay, event detectors are also asked for snapshots, and existing snapshots in the ordering buffer are replaced by new ones. Snapshot events are purged from the buffer like any other events.

Fig. 8 shows the ordering unit of Fig. 7 with snapshot processing. On top of each emitted grey event there is a special snapshot event in the buffer (denoted by s_{ts}) holding the state of the detector before the premature event was processed. Consider the replay situation at C1 in column 3. The snapshot $\underline{s}_2$ that has been taken when A2 has been emitted speculatively (column 2) is still in the buffer. For the replay, this snapshot is emitted first, followed by C1 and A2. The procedure is similar when C9 arrives.

If two subsequent snapshots do not differ, we just store a reference to the prior snapshot. Nevertheless speculation needs extra storage. The space needed grows with the degree of speculation and depends on the complexity of the event detector's state space.

Algs. 1 and 2 show how the speculation works. We use worker threads that iterate over the ordering units and that are used by the event detectors for event processing (Alg. 2). While such worker threads may be busy with processing events, a new event may be received, and Alg. 1 is called upon reception. For any event we receive, we first calculate its delay and update K by means of classic K-slack, if necessary. Note that as events may have been generated prematurely we have to add K^+, i.e., the premature emission time that we attached to the event, on top of the event's delay. Next, since this event may be out-of-order, Alg. 1 acquires the lock on the ordering unit's buffer, i.e., stops the event detector processing, inserts the (out-of-order) event at its correct position, reinitializes the event detector, and relocates the AEP, if needed. With its termination it triggers the worker thread to proceed. The worker threads are also triggered by clk updates, and are also used to purge the obsolete events from the buffer.

4.3 Event Retraction

If some event is missing in the speculatively emitted stream, we restore the snapshot of the subscribing event detector and replay the event stream. What remains open is that this event detector may itself already have generated events based on the incomplete event stream. These events must be retracted/eliminated from event streams subscribed by detectors on higher hierarchy levels. Since this may lead to a heavy retraction cascade we must limit the degree of speculation.

Consider the example event detector of Fig. 3(a) and the speculative buffer in Fig. 9(a). For this example we assume that the event detector numbers the generated D events. The event detector has already speculatively processed the grey events A3 to B8 and has already generated D_i3 out of A3/C5 and $D_{i+1}6$ out of A6/C7 when an event B with time stamp 4 arrives. Since the subscribing event detector itself has incorrectly generated D_i3, we must restore the event detector's state, replay C5 to B8, and retract D_i3 from the streams of higher level event detectors.

Moreover, $D_{i+1}6$ may be wrong as well because of two reasons. First, the event detector may not have reached the D-state in presence of B4, because of some internal state variables that are not shown. Second, even if it would reach the D-state then instead of $D_{i+1}6$ it should have produced D_i6. Hence, in addition to be able to replay event streams the middleware must also be ready to invalidate events that have been generated based on prematurely emitted events.

The key idea to restore a correct state at a higher level event detector H is to send a retraction event that identifies the events that have incorrectly generated so that H's ordering unit can fix that and replay the event stream.

Below we present two techniques to handle event retractions across the detection hierarchy: *Full Retraction* and *On-Demand Retraction*.

4.3.1 Full Retraction

The key idea of *Full Retraction* is to instantly retract all events that may have been generated incorrectly as soon as the AEP is relocated. For this purpose, an event detector's ordering buffer must not only store the prematurely emitted events and the snapshots but must conceptually also hold a list of events that the detector has generated from the prematurely emitted events, i.e., $\mathtt{D}_i\mathtt{3}$ and $\mathtt{D}_{i+1}\mathtt{6}$ in the example. When an out-of-order event is inserted into the buffer, we first collect all events that may have been incorrectly generated, and send a (conceptual) retraction event for each of them to the ordering unit of each subscribing higher level event detector H. When this ordering unit receives such a retraction event it purges this event from its buffer, and performs its own retraction and replay. Hence, a retraction event reuses the replay and snapshot recovery used for out-of-order events. For instance, in Fig. 9(a) we insert B4 between A3 and C5, instantly send retraction events for $\mathtt{-D}_i\mathtt{3}$ and $\mathtt{-D}_{i+1}\mathtt{6}$, tell the event detector to restore the appropriate snapshot, and start the replay.

Although retraction of every single incorrectly generated D event would work, there is a more efficient way to achieve the same effect. The idea is to exploit an event counter i that the ordering unit attaches to the detector's state as it has been done by the event detector before.[6] Instead of sending several retraction events ($\mathtt{-D}_i\mathtt{3}$ and $\mathtt{-D}_{i+1}\mathtt{6}$ in the example), it is sufficient just to send the event *counter* $\mathtt{D}{:}i\text{-}1$ to the upper level detector. This detector can then purge all D events from its buffer that have a larger counter.

But the event counter not only helps to reduce the number of retraction events that need to be sent to higher level detectors. With the counters stored in the states, there is no longer a need to keep lists to the generated events. In the example, there is no need to store the lists $\mathtt{C5\text{->}D}_i\mathtt{3}$ and $\mathtt{C7\text{->}D}_{i+1}\mathtt{6}$. Instead the counter values are piggybacked to the states s. This reduces the necessary footprints.

The advantage of full retraction is that the ordering units of higher level event detectors purge retracted events from their buffers and replay their event streams immediately. If the event detector's state changes and/or the prematurely generated events differ, full retraction works as efficient as possible. Full retraction is essentially the state-of-the-art in the related work.

4.3.2 On-Demand Retraction

With full retraction and its purging of generated events, the detectors have to perform their detection work again. This consumes CPU cycles. But consider state-less detectors that will generate exactly the purged events again. It is obvious that for those detectors most of the retraction work is wasted. The efficiency of full retraction and the achievable degree of speculation strongly depend on the internal structure of the event detector and its generated events.

The key idea of *on-demand retraction* is not to send the retraction events immediately upon AEP relocation. Instead we replay the event stream and only retract events if snapshots change and/or if events are not generated again during replay. In more detail, whenever we emit events during replay we check if the following two properties hold:

(1) Snapshots are equal. If we replay the event stream, and the snapshots do not differ we can abort the replay process. Because the upcoming premature events in the replay cause the same snapshots and hence generate the same events both the snapshots and the previously generated premature events remain valid. We assume that event detectors solely process events emitted by the middleware. Other input, e.g., system calls or file handles, are not allowed.

(2) Generated events are equal. The events and their counters that are generated again during replay are marked as updates, and the ordering unit of the higher level event detector H checks if the previously generated premature event equals the recently generated update event. If it does, the ordering unit of H does not reinsert the new event, does not relocate the AEP, and hence does not trigger an unnecessary retraction cascade.

Fig. 9(b) shows on-demand retraction for the example. When B4 is inserted into the buffer, the event detector is reset to use the snapshot $\underline{s}_4$=$\underline{s}_5$, and works on the replayed events. The event detector will reach some state $s_5{}'$ after processing B4 speculatively (not shown in Fig. 9(b)). If $s_5{}'$=$\underline{s}_5$(=$\underline{s}_4$), i.e., if the state of the event detector is not affected by B4, we can abort replay and retraction because all the subsequent snapshots and the prematurely generated events will remain unchanged.

However, if the state of the event detector is affected, we replay the event streams, and whenever an event is generated, we set the *update* flag before we send it to the ordering unit of H. The ordering unit of H then checks the updated event w.r.t. equality and discards it if there is no change.

If the event detector is state-less or events that cause a replay do not change much of the output, on-demand retraction considerably reduces retraction work that would be introduced by full retraction across the detector hierarchy. See also the evaluation in Sec. 5.

4.4 Runtime α-adaptation

Speculation uses additional system resources to reduce event detection latency. The remaining question is how to set the attenuation factor α that controls the degree of speculation. The ideal value of α results in best latencies but also avoids exhausting the available system resources and hence system failure.

We now present a runtime α-adaptation algorithm that achieves this goal. It is safe because when either no runtime measurements are available or when a critical situation occurs, e.g., a CPU load that is higher than some threshold, α is (re-)set to its initial value α_0=1 in order to prevent a system failure. Moreover, α-adaptation only has a local effect because α only effects replay and retraction of a single ordering unit. The CPU load on machines that run other detectors are not affected much because the speculative buffer of an upper level event detector only inserts and/or retracts events but does not start a replay (which in general depends on its own α).

The key idea of our runtime α-adaptation is to use a control loop similar to the congestion control mechanism known from the transmission control protocol (TCP) [27]. Congestion control tries to maximize throughput by doubling the data rate, i.e., the congestion window size that holds the number of to-be-acknowledged packets, at each time unit. When the data rate becomes too high for the link, data packets are timed out because packets are lost in the network, and the window size is reduced to 1. The maximal window size is saved and a new iteration begins. The window size is doubled again until it reaches half the window size of the previous iteration. Then its size is incremented until again packets are lost and the next iteration starts over.

[6] The time stamps cannot be used as event counter because the ordering middleware has no influence on their generation and they are not said to be increasing.

RIGHTSLINK

Algorithm 3: α-adaptation.

```
Data: $\alpha$, $\alpha_s$, lastMinimum, bool slowmode, $b_l$, $b_u$, $b_c$
begin
    if $b_u < b_c$ then                          // reduce speculation
        lastMinimum ← $\alpha$;
        $\alpha \leftarrow 1$;
        slowmode ← false;
    if $b_c < b_l$ then                          // increase speculation
        $\alpha_{target} \leftarrow 0$;
        if slowmode then
            $\alpha_{target} \leftarrow \alpha - \alpha_s$;
        else
            $\alpha_{target} \leftarrow \alpha/2$;
            if (1- lastMinimum) / 2 > $\alpha_{target}$ then
                slowmode ← true;    // goto slow mode
                $\alpha_{target} \leftarrow \alpha - \alpha_s$;
        $\alpha \leftarrow \alpha_{target}$;
```

To adapt that idea, we use α as the congestion window size, and the CPU workload as a load indicator. Whenever we evaluate the CPU load we adjust the value of α. To measure the CPU load accurately, the middleware repeatedly (after each time interval t_{span}) sums up the times that event detectors need for event processing, i.e., t_{busy}, and relates it to the sum of the times the worker threads have available, t_{span}. The resulting busy factor b_c is

$$b_c = 1 - \frac{t_{busy}}{t_{span}}.$$

For instance, with a time interval t_{span}=0.5s and an accumulated t_{busy}=0.41s the busy factor b_c is 0.41s/0.5s=0.82. This means that 82% of the available resources are used and that about 18% of the system resources are still available (assuming that no other processes interfere with the EBS). The busy factor grows with decreasing values of α.

To adjust α we specify a target zone $[b_l; b_u]$, i.e., an interval of the *lower* and the *upper* target values for b_c. If the busy factor is below b_l, CPU time is available and α is decreased by halving its value. This is similar to the doubling the congestion control window size. α is the inverse of the window size and halving α increases speculation. If the busy factor grows above b_u, CPU time becomes critical, and the current α is kept (called α_{best}) before α is set to its initial value α_0=1. From there α is again halved until its value is about to be set below the bisection line (1-α_{best})/2. From then on α is lowered in small steps of α_s, to slowly approach the target zone. Alg. 3 gives the pseudo code for the α-adaptation.

Our evaluation shows that the busy target interval $[0.8; 0.9]$ in combination with α_s=0.05 works reasonably well. Of course, b_c is not only affected by the choice of α. In burst situations or when a detector reaches a rare or slow area of its state space, t_{busy} may peak. In such cases, the runtime α-adaption reacts appropriately by resetting α and hence by giving more resources to the detector.

5. EVALUATION

For the evaluation we have analyzed position data streams from a Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. This RTLS tracks 144 transmitters at the same time at 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Each player has four transmitters, one at each of his limbs. The sensor data consists of absolute positions in millimeters, velocity, acceleration, and Quality of Location (QoL) for any direction [28].

Soccer needs these sampling rates. With 2,000 sampling points per second for the ball and a velocity of up to 150 km/h, two succeeding positions may be more than 2cm apart. Soccer events such as pass, double pass, or shot on goal happen within a fraction of a second. A low latency is required so that a hierarchy of detectors can help the human observer, for example a reporter, or a camera systems that should smoothly follow events of interest, to instantly work with the live output of the system.

We present results from applying our event processing system and our algorithms on position data streams from the stadium. Our platform consists of several 64-bit Linux machines, each equipped with two Intel Xeon E5560 Quad Core CPUs at 2.80 GHz and 64 GB of main memory that communicate over a 1 Gbit fully switched network. For our tests we organized a test game between two amateur league soccer clubs and processed the incoming position streams from the transmitters.[7]

For all benchmarks we replay position stream data in our lab's computing cluster. To focus on the experimental results more clearly, we perform all the work on just one machine. Our event processing middleware, i.e., the methods for speculative processing, pub/sub-management, etc., take around 9,000 lines of C++ code. On top of that we implemented over 70 event detectors with more than 16,000 lines of C++ code that are used to detect more than 750 different event types. The event detection hierarchy has 15 levels, and we replay a snippet of the event stream from the soccer match. The duration of the test event stream is 65 seconds and consists of 2.2 million position events plus 25,000 higher-level events that are generated by the event detectors (not including prematurely emitted events or retraction events). The data stream also incorporates some burst situations. The average data rate of the processed data streams is 2.67 MBytes/sec.

We let the event detectors work on the data streams, and discuss the generated results. For all our experiments we use only one worker thread to make the results more clear and to avoid side-effects. Sec. 5.1 shows that speculative processing can considerably reduce both latencies and detection delays. Sec. 5.2 evaluates the α-adaptive speculation in detail. Sec. 5.3 and 5.4 provide measurements on the resource consumption during the event stream replay versus full speculative and pure buffering approaches.

5.1 Latency reduction

Fig. 10 shows that added speculation can significantly reduce latency of buffering middlewares. For the benchmark we measured the latency of the *pass* event detector. This detector subscribes to 6 different event types and detects a (unsuccessful) *pass* event.

When we replay the event data stream to a pure dynamic K-slack buffering (α=1, straight line) it updates K upon ordering mistakes and finally ends up with a detection latency of 1458ms at the end of the stream replay. The average latency for the 65 seconds is 1276ms. In contrast, our

[7] FIFA rules do not allow a continuous operation in premier league matches.

RIGHTSLINK

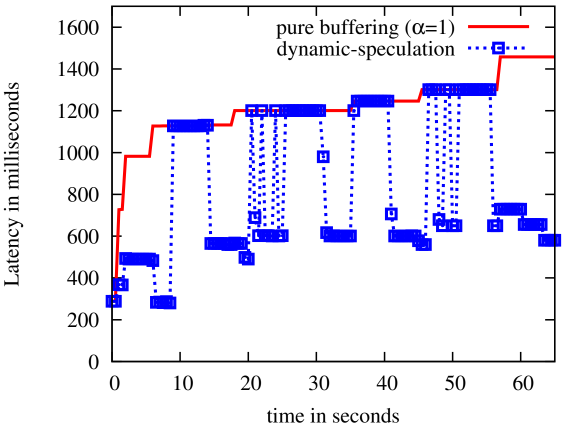


Figure 10: Latencies of (speculative) buffering.

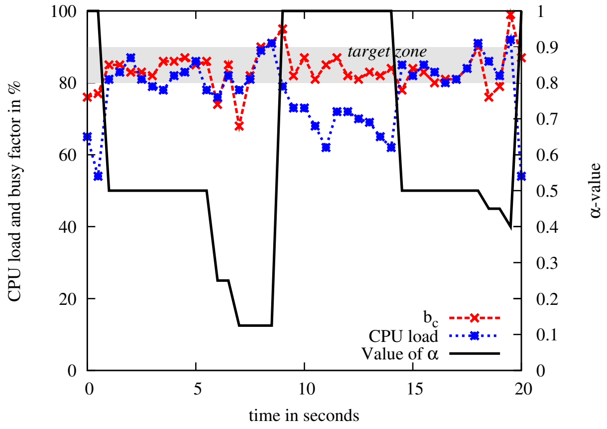


Figure 11: Adaptation of α.

dynamic α-adaptation reaches a much smaller detection latency (dotted line). At the beginning, α starts at α_0=1 and around 290ms of latency. As the CPU is not fully loaded α-adaptation switches on speculation. This brings detection latency down to 280ms, where pure buffering already has 1128 ms latency. Note that the latency of pure buffering increases much more than that of dynamic speculation as α-adaptation outweighs the growth of the K-slack buffer. At several points the event streams, and hence the busy factor burst. That causes α to back off, leading to a higher detection latency again. Afterwards, α-halving resumes and the latency approaches its minimum again. The average latency of the dynamic speculation is below 800ms, i.e., about 40% better than what pure buffering can achieve.

These results show that our speculative buffering technique strongly reduces the detection latency of events at runtime. Throughout the entire 65 seconds the CPU load was tolerable, and at the critical burst situations α-adaptation can avoid system failures, see Sec. 5.2. Hence, camera movements can be triggered about 40% faster than with pure buffering techniques. The latencies of other event detectors behave similarly.

We did not show the latency of full speculation (α=0) because it consumes too much CPU power and causes event processing to fail. See Sec. 5.3 for more details. Other variants with static α-values would just result in latencies that represent the original latency divided by the static α-value.

5.2 Runtime α-adaptation

In the above measurements there are several bursts where α has to back off due to high system loads. To discuss the performance of our α-adaptation in more detail, let us zoom into the first 20 seconds of the event stream, see Fig. 11.

With a busy factor target zone $[0.8; 0.9]$, we start from α=1 (straight line). We evaluate and (possibly) halve α every t_{span}=0.5 seconds, and as a result the busy factor b_c (dashed line) grows. After 7 seconds α is 0.125, we cease halving, and b_c stays in its target zone between 0.8 and 0.9. For a while, both the busy factor b_c and the CPU load fluctuate within the target zone before a growing frequency of incoming events requires more and more detection work. After 8.5 seconds into the benchmark b_c reaches 0.95 (the CPU load at that time was at 91%), α is immediately reset to 1, and as a result both b_c and the CPU load drop instantly. Then α is evaluated and after 16 seconds α-halving starts over. But this time halving stops at the bisection line (1.0-0.125)/2=0.43. From there α takes small steps of 0.05 per t_{span} to bring b_c and CPU load back into the target zone.

The α-adaptation algorithm not only decreases α to efficiently use the available CPU power but also rapidly stops speculation if the system is almost overloaded in burst situations, to avoid system failure and to absorb the bursts. Moreover, b_c is a sufficiently good indicator of the CPU load. Only if the CPU load is low, b_c slightly overestimates because of task switches and thread sleeping.

5.3 Resource Consumption

To measure the resource consumption we replay the event stream snippet three times. Fig. 12 shows the resulting CPU loads. We recorded the CPU loads for pure K-slack buffering (α=1, straight line), dynamic α-adaptation (dotted line), and fully speculative processing (α=0, dashed line).

Pure buffering exhibits a comparatively lazy CPU load of 50-70% for the entire 65 seconds. That is because events are buffered for a sufficiently long time and the event detectors neither receive retraction events nor are they asked for snapshots or get restored. In contrast, full speculation causes a high CPU consumption above 90% for the entire 65 seconds. In the benchmark the CPU consumption reaches 100% a few times. This is prohibitive because event detection then takes longer than it should. The resulting higher delays cause a ripple effect since the K-values of the detectors further up the detection hierarchy are not prepared to deal with the extra delay and purge events or process them out-of-order. Hence, detectors may get stuck in invalid states and event processing fails. Even worse, there is another reason for system failure: when event processing is slowed down, the queue of incoming events that await being processed often outgrows the available buffer space.

Fig. 12 also shows that to achieve the good detection latencies discussed in Sec. 5.1 the dynamic speculation makes reasonable and efficient use of the available resources. The CPU load stays within a non-critical target zone that gives the dynamic α-adaptation enough room to react to bursts and to avoid system failures. Hence, if detection latencies in an application are too high, one simply needs to use a stronger CPU. Then there is more room for speculation and hence latency reduction.

Speculation also needs more main memory. On our benchmark, pure buffering took only 1,120 KBytes of buffer space, averaged over the 65 seconds. There are two reasons for the demand being that small. First, only the events but not the

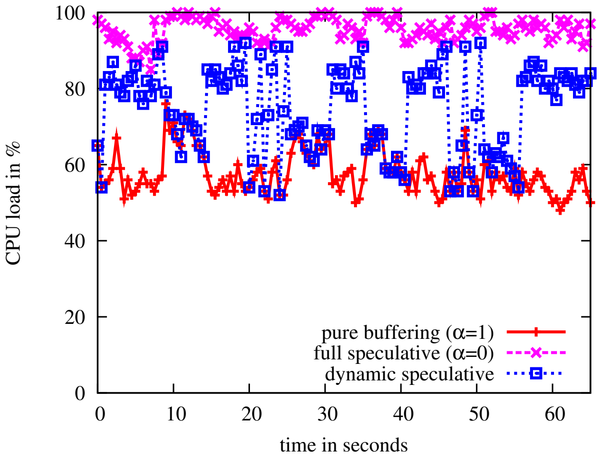


Figure 12: CPU loads with varying α.

detectors' snapshots need to be stored. And second, since an event is usually input to more than one detector we store it only once and insert a reference to it into the ordering units. In contrast, as an event detector's state has an average size of around 800 Bytes, full speculation required 21,100 KBytes for an event detector's buffer on average. Dynamic speculation is better and only took 14,850 KBytes on average, i.e., with only 14 MBytes of additional main memory per detector one can buy a latency reduction of 40%.

States of event detectors from the lower levels are considerably smaller (below 50 bytes) than the states from higher level detectors (one unique detector has a state size of 800 KBytes). Beyond that, event detectors of lower levels are snapshot more frequently than higher level event detectors since the event load is considerably higher.

5.4 Evaluation of Retraction Techniques

To evaluate the two retraction techniques we replay the real event stream from the first half of the soccer match (45 minutes). For each of the two retraction techniques we record the number of events that are prematurely generated and retracted from a *player-hits-ball* event detector.

On-demand retraction is much more light-weighted and in general also helps reduce latency. Over the 45 minutes, full retraction affected 5,623 ball hit events, whereas on-demand retraction only had to work on 735 events. With full retraction, the resulting retraction cascade affected 117,600 events across the detector hierarchy, compared to 12,300 events for on-demand retraction. To explain the advantages of on-demand retraction, more than just the event counts need to be considered. Compared to full retraction it is more costly for the on-demand retraction to process snapshots and to check for state changes. More work takes more CPU power away so that α-adaptation selects a lower degree of speculation, which results in a higher latency. On the other hand, the many retraction events of full retraction are quicker to deal with – but there are so many of them.

On-demand retraction works best for event detectors with small states or with states that do not change much (or not for every single event). For example, event detectors that directly process sensor data or low-level events. For such detectors, on-demand retraction can (1) abort the retraction cascade from the bottom of the event hierarchy, and (2) check detector state changes quickly because snapshots are small or not even necessary.

But there are rare event detectors that work better with full-retraction. The *pass* detector is an example. First, it changes its state on almost every event received. Thus, all its checking of snapshots and of generated events is useless work. Second, full retraction only affects 819 events. On-demand retraction can only reduce that number by 27 which is not enough to amortize its cost.

In total, choosing on-demand retraction *for all detectors* is better than full retraction. In the benchmark averaged over all detectors the α-value of full retraction was 0.81 whereas on-demand retraction only achieved an average α-value of 0.69. Nevertheless, on-demand retraction achieved a 15% better detection latency than full retraction. It is future research and an optimization problem of its own to automatically deduce and assign the best retraction technique *for each individual detector* at runtime. That might improve performance even more.

5.5 Discussion

We first showed that our speculative buffer with dynamic α-adaptation reduces latency considerably. We zoomed into the details and proved that our method exploits available system resources progressively until we reach a predefined level. Comparing the resource consumption of our approach to full-speculative as well as buffering approaches shows that we achieve a perfect trade-off between latency reduction and resource consumption.

We deliberately did not present results on the efficiency of taking and restoring snapshots. While memory consumption has shown to be moderate but tolerable, the time for taking snapshots was too small in our benchmarks to matter at all. Hence, there is also no need for more sophisticated mechanisms like transactional memories etc. However, depending on both the event detectors' states and the events, this may be an option for improvement for other applications.

6. CONCLUSION

Our speculative buffer extension achieves reliable and low-latency event processing under the predominance of out-of-order events. Any buffering middleware can use our speculation to process buffered events earlier and hence to reduce detection latency by exploiting available CPU and memory resources whereas conservative buffering approaches can natively not use them. Our speculation needs no a-priori knowledge of event delays nor the internal description of the event detectors and the system adaptively adjusts the degree of speculation at runtime.

An evaluation of the presented methods on position data stream from a Realtime Locating Systems (RTLS) in a soccer application shows that our dynamic speculation outperforms other speculative and buffering techniques. On average, we achieved a 40% reduction of latency.

Future work will refine the α-adaptation to incorporate event loads and latencies per event detector more specifically. Moreover, we will investigate approaches that automatically deduce and select the best retraction technique for each individual event detector at runtime.

Acknowledgements

This work is supported by the Fraunhofer Institute for Integrated Circuits IIS whose RTLS called RedFIR we have used. We are grateful to the researchers at the Fraunhofer IIS for sharing their work and system with us.

7. REFERENCES

[1] M. Stonebraker, U. Çetintemel, and S. Zdonik, “The 8 requirements of real-time stream processing,” *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, 2005.

[2] D. O’Keeffe and J. Bacon, “Reliable complex event detection for pervasive computing,” in *Proc. 4th Intl. Conf. Distributed Event-Based Systems*, (Cambridge, UK), pp. 73–84, 2010.

[3] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, “Exploiting punctuation semantics in continuous data streams,” *IEEE Trans. Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[4] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, “Out-of-order processing: a new architecture for high-performance stream systems,” in *Proc. VLDB Endow.*, vol. 1, (Auckland, NZ), pp. 274–288, 2008.

[5] U. Srivastava and J. Widom, “Flexible time management in data stream systems,” in *Proc. 23rd ACM Symp. Principles Database Systems*, (Paris, France), pp. 263–274, 2004.

[6] E. Wu, Y. Diao, and S. Rizvi, “High-performance complex event processing over streams,” in *Proc. ACM Intl. Conf. Management of Data*, (Chicago, IL), pp. 407–418, 2006.

[7] M. Li, M. Liu, L. Ding, E. Rundensteiner, and M. Mani, “Event stream processing with out-of-order data arrival,” in *Proc. 27th Intl. Conf. Distrib. Comp. Systems Workshops*, (Toronto, CAN), pp. 67–74, 2007.

[8] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, “Efficient pattern matching over event streams,” in *Proc. ACM Intl. Conf. Management of Data*, (Vancouver, CAN), pp. 147–160, 2008.

[9] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, “Towards expressive publish/subscribe systems,” in *Proc. 10th Intl. Conf. Extending Database Technology*, (Munich, Germany), pp. 627–644, 2006.

[10] A. Demers, J. Gehrke, B. Panda, M. Riedewald, V. Sharma, and W. White, “Cayuga: a general purpose event monitoring system,” in *Proc. 3rd Biennial Conf. Innovative Data Systems Research*, (Pacific Grove, CA), pp. 412–422, 2007.

[11] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, “Cayuga: a high-performance event processing engine,” in *Proc. ACM Intl. Conf. Management of Data*, (Beijing, China), pp. 1100–1102, 2007.

[12] L. Brenna, J. Gehrke, M. Hong, and D. Johansen, “Distributed event stream processing with non-deterministic finite automata,” in *Proc. 3rd Intl. Conf. Distributed Event-Based Systems*, (Nashville, TN), pp. 3:1–3:12, 2009.

[13] M. Balazinska, Y. Kwon, N. Kuchta, and D. Lee, “Moirae: history-enhanced monitoring,” in *Proc. 3rd Biennial Conf. Innovative Data Systems Research*, (Pacific Grove, CA), pp. 375–386, 2007.

[14] A. S. Maskey and M. Cherniack, “Replay-based approaches to revision processing in stream query engines,” in *Proc. 2nd Intl. Workshop Scalable Stream Processing Systems*, (Nantes, France), pp. 3–12, 2008.

[15] D. Anicic, S. Rudolph, P. Fodor, and N. Stojanovic, “Retractable complex event processing and stream reasoning,” in *Proc. 5th Intl. Conf. Rule-based Reasoning, Programming, and Applications*, (Fort Lauderdale, FL), pp. 122–137, 2011.

[16] C.-W. Li, Y. Gu, G. Yu, and B. Hong, “Aggressive complex event processing with confidence over out-of-order streams,” *Comp. Science and Technol.*, vol. 26, no. 4, pp. 685–696, 2011.

[17] E. Ryvkina, A. S. Maskey, M. Cherniack, and S. Zdonik, “Revision processing in a stream processing engine: a high-level design,” in *Proc. 22nd Intl. Conf. Data Engineering*, (Atlanta, GA), pp. 141–143, 2006.

[18] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber, “Speculative out-of-order event processing with software transaction memory,” in *Proc. 2nd Intl. Conf. Distributed Event-Based Systems*, (Rome, Italy), pp. 265–275, 2008.

[19] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov, “Tolerating latency in replicated state machines through client speculation,” in *Proc. 6th USENIX Symp. Networked Systems Design and Implementation*, (Boston, MA), pp. 245–260, 2009.

[20] M. Liu, M. Li, D. Golovnya, E. Rundensteiner, and K. Claypool, “Sequence pattern query processing over out-of-order event streams,” in *Proc. 25th Intl. Conf. Data Eng.*, (Shanghai, China), pp. 784–795, 2009.

[21] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, “Consistent streaming through time: a vision for event stream processing,” in *Proc. 3rd Biennial Conf. Innovative Data Systems Research*, (Pacific Grove, CA), pp. 363–374, 2007.

[22] B. Chandramouli, J. Goldstein, and D. Maier, “High-performance dynamic pattern matching over disordered streams,” in *Proc. VLDB Endow.*, vol. 3, (Singapore), pp. 220–231, 2010.

[23] Z. Jerzak and C. Fetzer, “BFSiena: a communication substrate for StreamMine,” in *Proc. 2nd Intl. Conf. Distributed Event-Based Systems*, (Rome, Italy), pp. 321–324, 2008.

[24] G. Mühl, L. Fiege, and P. Pietzuch, *Distributed event-based systems*. Springer, Berlin, 2006.

[25] S. Babu, U. Srivastava, and J. Widom, “Exploiting k-constraints to reduce memory overhead in continuous queries over data streams,” *ACM Trans. Database Systems*, vol. 29, no. 3, pp. 545–580, 2004.

[26] C. Mutschler and M. Philippsen, “Distributed low-latency out-of-order event processing for high data rate sensor streams,” in *Proc. 27th Intl. Conf. Parallel and Distributed Processing Symposium*, (Boston, MA), pp. 1133–1144, 2013.

[27] M. Allman, C. Hayes, and S. Ostermann, “An evaluation of TCP with larger initial windows,” *SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 3, pp. 41–52, 1998.

[28] T. v. d. Grün, N. Franke, D. Wolf, N. Witt, and A. Eidloth, “A real-time tracking system for football match and training analysis,” in *Microelectronic Systems*, pp. 199–212, Springer Berlin, 2011.

RIGHTSLINK