Dynamic Low-Latency Distributed Event Processing of Sensor Data Streams

Christopher Mutschler[1,2] and Michael Philippsen[1]
[1] Programming Systems Group, Computer Science Department
University of Erlangen-Nuremberg, Germany
[2] Sensor Fusion and Event Processing Group, Locating and Comm. Systems Dept.
Fraunhofer Institute for Integrated Circuits IIS, Erlangen, Germany
{christopher.mutschler;michael.philippsen}@fau.de

Abstract: Event-based systems (EBS) are used to detect meaningful events with low latency in surveillance, sports, finances, etc. However, with rising data and event rates and with correlations among these events, processing can no longer be sequential but it needs to be distributed. However, naively distributing existing approaches not only cause failures as their order-less processing of events cannot deal with the ubiquity of out-of-order event arrival. It is also hard to achieve a minimal detection latency.
This paper illustrates the combination of our building blocks towards a scalable publish/subscribe-based EBS that analyzes high data rate sensor streams with low latency: a parameter calibration to put out-of-order events in order without a-priori knowledge on event delays, a runtime migration of event detectors across system resources, and an online optimization algorithm that uses migration to improve performance.
We evaluate our EBS and its building blocks on position data streams from a Realtime Locating System in a sports application.

1 Introduction

High data rate sensor streams occur in surveillance, finances, RFID-systems, click stream analysis [BBD+02], sports [GFW+11], etc. Event-based systems (EBS) turn the high data load into smaller streams of meaningful events, filter, aggregate, and transform them into higher level events until they reach a granularity appropriate for an end user application or for triggering some action. EBS are as manifold as their requirements with respect to workloads and response times. For instance, a warehouse management system must focus on the interface against which users program detection rules. It must hence interpret queries at runtime. In contrast, EBS on distributed sensor nodes must avoid communication to save energy and must filter and aggregate early as memory is only available sparsely.
This paper addresses EBS that also process data from a high number of sensors but that have sufficient CPU power and main memory, as for instance in finance, autonomous camera control (e.g. sports), or safety critical systems. When distributed processing of event detection can be made fast enough, actions can be triggered with low latency. Consider the soccer event dependency graph to detect a shot on goal from position sensor data provided by a Realtime Locating System (RTLS), see Fig. 1. Since even the detection of proximity in layer 4 (depending on the number of tracked transmitters) already consumes a machine's processing power, sub-event detectors must be spread to other nodes.

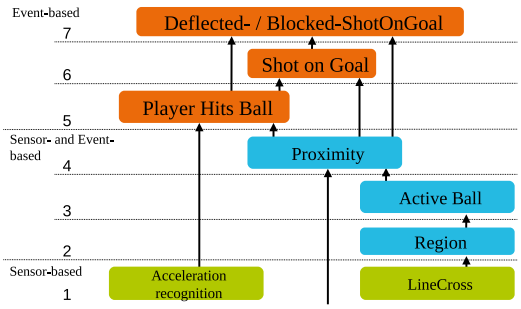


Figure 1: Event processing hierarchy.

Distribution of EDs across several machines is difficult. As events are generated at different network points and are no longer timely synchronized, delays and out-of-order events are predominant.[1] Thus, distributed EDs process events incorrectly. As a-priori defined parameters for event ordering do not help, we derive low-latency ordering parameters at runtime, see Sec. 4.1. For any predefined allocation of EDs to machines, unexpected events (stock trades, moves in a sports game, etc.) easily cause load imbalances and exhausted CPUs. Sec. 4.2 presents a technique to migrate an overloaded ED at runtime to another node, carefully considering the challenging timing delays.

Often there are hundreds of EDs linked by event dependencies and with a-priori unknown loads. But in order to trigger actions, e.g., smooth camera movements, by events the detection latency must be minimal. A poor allocation of event detectors (EDs) over available nodes may cause high detection latencies that are impractical to automatically trigger actions. An optimal allocation of EDs to nodes can only be found at runtime. As there are no polynomial time solutions and as greedy approaches often end up in local optima, Sec. 4.3 presents a latency-minimizing allocation heuristic.

We discuss related work in Sec. 2, sketch the architecture of our system in Sec. 3, and present the technical contributions of our EBS in detail in Sec. 4. Our techniques work well on position streams from an RTLS in a soccer application, see Sec. 5.

2 Related Work

The Borealis Stream Processing Engine [AAB+05] is the only other EBS we know that also combines stream revision processing, optimization, and fault avoidance [HXZ07, TcZ07]. Its stream revisioning by Cerniack et al. [CBB+03] uses so-called time-travels and processes the events that are buffered when revisions occur. However, for predominant out-of-order events almost all generated events must be revised, often triggering a cascade of revisions along the detector hierarchy. As this puts pressure on the memory management, Borealis limits the memory for revision processing and drops packets that cannot be revised to implement an inherent load shedding. Of course, the resulting corrupt states can make the EDs fail. In contrast, we dynamically determine the minimal buffer sizes that avoid such failures and revision cascades. For the details of our out-of-order event processing and a comparison to specific related work, see [MP13b].

Whereas the Borealis box-sliding operator migration from the same paper [CBB+03] can only move EDs to the lower level ED's node or to the higher level ED's node (because it is not order-preserving), we present a technique that can migrate EDs between arbitrary

[1] Delays are not only introduced by network or CPU. Events that are generated with earlier time-stamps than the time-stamps of the events that cause them have a detection delay and can only be inserted into the event stream long after they have actually happened. A technique for event ordering must also address this delay type.

nodes, independent from the detector hierarchy. More details and a more thorough discussion of related work in the area of migration can be found in [MP13a].
Xing et al. [XZH05] add dynamic load distribution to Borealis. Their greedy approach migrates filter operators pair-wise and minimizes end-to-end latency by lowering load variance and by increasing load correlations. They periodically collect CPU load statistics and either a one- or two-way algorithm migrates the operators. Whereas their greedy approach does not consider network latencies but only CPU loads and is likely to just optimize towards a local optimum, our technique migrates several EDs at a time and obtains a significant benefit in performance that is closer to the global optimum.
Pietzuch et al. [PLS+06] take a-priori statistics of event loads and calculate an optimal allocation of EDs in the network. Their method cannot be applied since usually event loads are unknown in applications that analyze highly dynamical data. Threshold-based techniques [YCJ06] optimize availability by monitoring network, CPU, and memory loads but do not particularly care about detection latency or bandwidth minimization.

3 Architecture

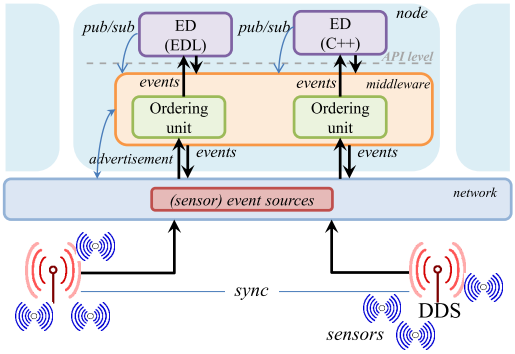


Figure 2: Distributed publish/subscribe EBS.

Fig. 2 depicts our distributed publish/subscribe-based event processing system. It is a network of several machines that run the same middleware to process sensor readings that are collected by a number of data distribution services (DDS), e.g., antennas that collect RFID readings. EDs are spread across the machines. An ED communicates subscriptions, publications and control information with the middleware that does not know the ED's event pattern; the ED is unaware of both the distribution of other EDs and the runtime configuration. The middleware implements a push-system with unknown subscribers. At system startup the middleware has no clue about event delays on other hosts but just notifies other middleware instances about event publications.
As it is difficult to manually implement EDs that process out-of-order events and developers often do not know the delays that their code may face at runtime, the middleware provides a personal event ordering unit per ED. For that, it extracts a local clock out of the event stream, see Sec. 4.1. The middleware is thus generic and encapsulated, and does not incorporate the application-specific event definition of the EDs.
In our application domains all system units that directly communicate with sensors are synchronized. Therefore there is a way to time-stamp sensor events with a single discrete time source at the point where they are generated.[2]

[2]In warehouse applications, the RFID-readers may synchronize over LAN, time-stamp the sensor readings accordingly, and push the data packets as sensor events to the network. In locating systems, the microwave signals of transmitters are extracted by several antennas that are synchronized over fiber optic cables [GFW+11].

4 Building Blocks

We use the following terminology throughout the rest of the paper:

Event type, instance, and time-stamps. An event type is identified by a unique `ID`. An event instance is an instantaneous occurrence of an event type at a point in time. An event has two time-stamps: an occurrence and an arrival, both are in the same discrete time domain. An event appears at its occurrence time-stamp ts, or just time-stamp for short. At arrival time-stamp ats the event is received by a particular node. The occurrence time-stamp is fixed for an event at any receiver whereas the arrival time-stamps may vary.

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 the two consecutive clock updates.

Event stream. The input of an ED is a potentially infinite event stream that usually is a subset of all events, holds at least the event types of interest for that ED, and may include some irrelevant events as well.

4.1 Self-Adaptive Event Ordering

Our dynamic event ordering builds on K-Slack [LLD+07]. K-Slack assumes that an event e_i can be delayed for at most K time units. For a particular ED the ordering unit that turns a stream with potential out-of-order events into a sorted event stream uses the maximal delay K of all subscribed events to configure the size of a buffer for event ordering. The ordering unit is mounted between the event input stream and the ED, extracts a local clock clk from the event stream, and delays early events as long as necessary.

While there are EBS that use programmer-configured K-values [LLD+07], we have shown in [MP13b] that it is both possible and better to measure event delays at runtime and to configure optimal K-values dynamically. One reason is that EDs in practice often form a detection hierarchy with an ordering unit per ED so that latencies add up along the hierarchy. For instance, to automatically trigger smooth camera movements with events, all the K's in the hierarchy must be as small as possible (to steer the camera as early as possible), and must be as large as necessary to detect all the necessary events in the detection chain. Overly large K's result in large buffers and high latencies for EDs further up the hierarchy.

Fig. 3 shows how such an ordering unit works that is initialized with K=0. The ED has subscribed to three events `A`, `B`, and `C`. The ordering unit is mounted between the event input stream and the ED. It sets its local clock clk whenever it receives events (in this example we just take `A` to set clk). Whenever clk is updated we esti-

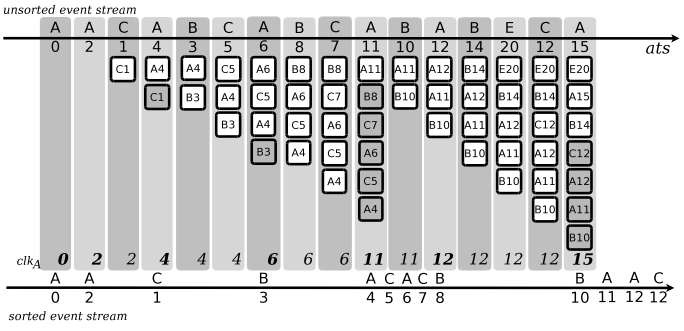
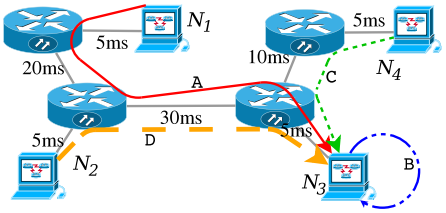


Figure 3: Event ordering unit for an ED.

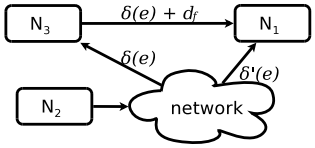
mate the maximal delay of all events that have been received since the last clk update, and retro-fit K if necessary. All events that are currently buffered are evaluated and emitted if they satisfy $e_i.ts+K \leq clk$. At the beginning, when `A0` and `A2` are received, K is still 0, which means that both events are immediately passed to the output stream. When `C1` is received, it is pushed to the buffer and waits until `A4` updates clk. As the delay for `C1` is 3=4-1, we set K=3 and relay `C1`. `A4` is buffered at least until clk equals 7. With `A6` the maximal delay of `B3` is 3, K holds, and `B3` is passed to the ED.
Hence, our method iteratively calibrates K and delays both late and early events as long as necessary to avoid out-of-order events. We use a λ scaling factor to overestimate K in relation to the standard deviation of the event delays to avoid sudden delay changes. We signal increases of K to upper level EDs by sending a pseudo event that passes the new delay to their ordering units. Two methods to initialize K at runtime so that there is no initial stumbling in the ordering units can be found in [MP13b]. We discuss space complexity and give results on the induced latency in a real-world setup in Sec. 5.1.

4.2 Runtime Event Detector Migration

A distributed EBS must somehow spread its EDs over the available system resources. However, the system behavior can hardly be estimated before runtime and a predefined allocation may perform poorly. An allocation may even be so poor that event processing may fail due to network or CPU overload. For instance, think of a rare event that triggers a cascade in the ED hierarchy for which not much computing power has been pre-allocated. Detection and event generation then takes considerably longer, causing even more out-of-order events at higher hierarchy levels. Runtime ED migration can help and also optimize the detection latency to trigger certain actions, see Sec. 4.3.
However, to migrate an ED from one node to another at runtime, not only requires to send the ED's state but also the measured event delays and the size of the reordering buffer, both of which are affected by the migration. Consider Fig. 4(a). An ED on host N_3 subscribes to four events: `A` (with 60ms delay), `B` (with 0ms delay), `C` (with 20ms delay), and `D` (with 40ms delay). If a particular delay increases after migration, the ED fails because the event ordering uses the original K so that the emitted out-of-order events make the ED fail. For instance, after migrating the ED to N_1 `B`'s delay grows by 60ms, there is a plus of 50ms for `C`, a minus of 10ms for `D`, and `A` becomes a local event without delay. Hence, the previous delays that configured the K of the ED's ordering unit are no longer valid on N_1. Related work from the areas of virtual machine migration and cellphone handover is insufficient or introduces long-lasting network or processing overheads [MP13a].



(a) Example topology.



(b) Delay $\delta(e)$ of event `e` before migration, $\delta'(e)$ after migration; d_f is the forwarding sub-delay.

Figure 4: Runtime migration of EDs.

The key idea of our runtime migration is that the new ED on the host N_1 not only subscribes the necessary input events from their sources, but the old host N_3 also forwards them to N_1 for a short while, see Fig. 4(b). N_1 can then derive the correct order by combining delay information from events that arrive along two paths and by taking into account the forwarding sub-delay δ_f. When migration is initiated, N_3 and N_1 agree on a moment, when the ED on N_1 takes over and N_3 deactivates its ED. Whenever N_1 measures an event delay of a directly received event, it can update its K-value (if necessary/possible) and signal N_3 to stop forwarding that event type. Echoed events are dropped. If the K-value for the migrated ED changes, we send a pseudo event to the ordering units of upper level EDs, so that they can adapt their K-value also. More details can be found in [MP13a].

4.3 Runtime Latency Optimization

Our runtime latency optimization moves EDs to improve latencies and, as a consequence, to also optimize their distribution so that they reduce the ordering unit sizes. As event streams cannot be predicted sufficiently well, plan-based approaches do not work for online optimization. As greedy approaches often end up in local optima we apply heuristics to optimize the ED distribution at runtime. This requires continuous monitoring of all event stream statistics in a centralized optimization master (OM) component. The OM collects the number of transmitted events for each `ID`, the measured network latencies to other nodes, and the number of generated events.

The OM periodically triggers a Cuckoo Search (CS) combined with a particle swarm optimizer (PSO) [GL12]. CS models the aggressive breeding behavior of cuckoos that lay their eggs (an ED distribution) into other birds' nests (a container to store distribution), and evicts the original eggs (selecting the fitter ED distribution). The host birds either breed the eggs or abandon the nest.

The key idea of the algorithm is that in each iteration $i \leq n$, a cuckoo performs either a lévy flight, i.e., it creates a new solution far away from the current solution (we use a Levensthein distance between ED distributions), or an ordinary low-distance movement depending on the solutions found recently. Next, the cuckoo randomly chooses a host nest for its egg, i.e., a container to store its distribution. The number of nests is fixed from the beginning. If the chosen nest is not empty, the fitness of both solutions/eggs is evaluated and the one with lower quality is evicted. The number of nest determines the number of concurrently existing solutions. The number of cuckoos configures how many new solutions are generated per iteration. After each iteration, a number of (bad) nests is replaced by new, empty ones. The lévy flight can be parameterized to control the variance of solution space scanning. By combining CS and PSO we replace the ordinary movement with a swarm behavior. The particular particles, i.e., the cuckoos, know each other, and adapt their lévy flights to reach areas with better solutions. This significantly improves the results.

Problem formalization. Let n be the number of EDs, m the node count. An allocation is a three-tuple consisting of an $m \times n$ matrix X with $x_{i,j}$=1 if ED i runs on node j, an $n \times n$ matrix L where $l_{i,j}$ is the latency between hosts i and j, and an $m \times m$ matrix T where $t_{i,j}$ is the number of events transmitted from node i to j. The goal is to minimize

$$\varphi\left(X, L, T\right) = \sum_{i=1}^{m}\sum_{j=1}^{m}\sum_{k=1}^{n} x_{i,k} \cdot x_{j,k} \cdot \left(c_t \cdot t_{i,j} + c_l \cdot l_{i,j}\right),$$

which describes the cost of the current ED distribution under the runtime measurements L and T, weighted by c_t (emphasis on network transmissions) and c_l (emphasis on latency).

Solution. An optimization based on heuristics needs (1) a formal encoding of possible solutions (individuums), and (2) a fitness function that grades the solution's performance.

Individuum. In an ED allocation $\vec{X} = (x_1, x_2, \cdots, x_n)^T$, with $x_i \in [1; m]$, an ED i runs on host x_i.

Fitness function. The fitness function is used to grade possible solutions and must project them onto numbers. The higher the quality $f(\vec{X})$ of a solution $\vec{X}$, the fitter is the individuum in the evolutionary algorithm. Our function incorporates network latencies and traffic, and is again weighted by the parameters c_t and c_l:

$$f(\vec{X}) = \sum_{i=1}^{n}\sum_{j=1}^{n} a_{i,j} \cdot c_t \cdot t_{i,j} - \sum_{i=1}^{n}\sum_{j=1}^{n} b_{i,j} \cdot \left(c_t \cdot t_{i,j} + c_l \cdot l_{i,j}\right).$$

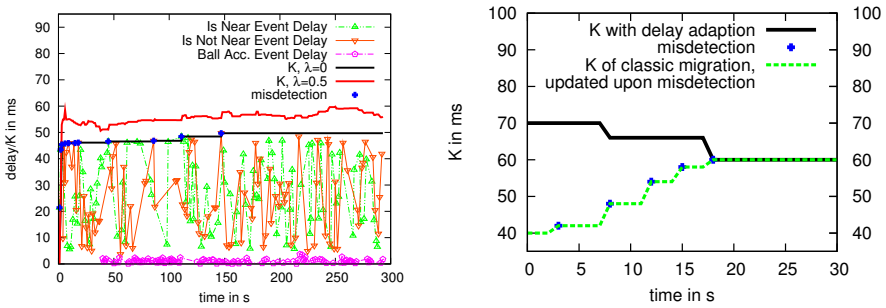
$a_{i,j}$ is 1 if EDs i and j run on the same host, i.e., $x_i = x_j$, and 0 otherwise. Contrary, $b_{i,j}$ is 1 if EDs i and j run on different hosts, and 0 otherwise. The left part of the fitness function sums up the *cost savings* by events that are transmitted locally between EDs and that must not travel through the network. The right part sums up the cost for ED dependencies that cause network traffic. Hence, $f(\vec{X})$ grows with locally transmitted events and decreases with events that are subscribed by hosts with higher latency or with more generated events. Due to limited CPU power we skip and discard solution vectors that would overload hosts.

5 Performance Results

We have analyzed position data streams from a Realtime Locating System (RTLS) installed in the main soccer stadium in Nuremberg, Germany. The RTLS tracks up to 144 transmitters at 2,000 sampling points per second for the ball and 200 sampling points per second for players and referees. Players are equipped with four transmitters, one at each of their limbs. The sensor data consists of a time-stamp, absolute positions in millimeters, velocity, acceleration, and Quality of Location (QoL) for any direction [GFW+11].
Soccer needs this 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. Hence, reducing the sampling frequency of position events would make detection of several events impossible. Since soccer events like pass, double pass, or shot on goal happen within a fraction of a second, event processing must ensure that events are detected in time so that a hierarchy of EDs can, for instance, control an autonomous camera system for smooth camera movements or help a reporter to work with the live output of the system.

5.1 Self-Adaptive Event Ordering

To evaluate the event ordering units we recorded the delays of incoming events for the *Player Hits Ball* ED, see Fig. 5(a). These events are *Is Near*, *Is Not Near*, both oscillating between 5 and 45ms delay, and *Ball Acceleration Changed* with a delay of 1-2ms. With



(a) Safety Margin and K.

(b) Delay adaption after migration.

Figure 5: K-values over time.

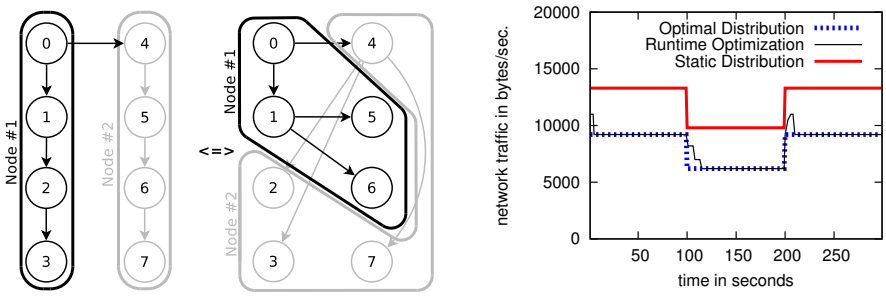
our self-adaptive dynamic K configuration, over 95% of all events are correctly ordered without a-priori knowledge, see the λ=0 line. There are rare points at which K is too small and causes a misdetection, see the mismatch crosses, before it is increased. With an overfitted K (added safety margin λ=0.5) not a single K misses the maximal event delay, and the ED is always supplied with ordered events. This ED detects ball hits with a delay of only 59.8ms whereas a manual K-selection by an expert would cause a latency >500ms. Hence, with our method smooth camera movements can be triggered early and automatically by events. The memory consumption of our buffering approach is negligible, since usually only a few milliseconds of events are buffered. The main result of our approach is to reduce the detection latency.

5.2 Runtime Event Detector Migration

To evaluate the runtime migration we replayed recorded test match data and processed it in our lab (an ESXi server with a cluster of VMs, each with a 2 GHz Dual Core CPU, 2GB of main memory, and 1 GBit virtual network communication configured to simulate a real environment). We migrated an ED that subscribes to four different event types and emits the *pass* event. We compared it to classic migration approaches. They either fail, are too slow, or take too many resources.

Fig. 5(b) shows the K-value of the ED migrated in the classic way (dotted line, K starts with 40ms). This ED fails 5 times within the first 17 seconds after migration, and whenever it fails, K is increased to prevent future misdetections (as dynamic K-slack would do). In contrast our migration technique lets the migrated ED start with a slightly larger K and lowers it whenever an event shows up earlier along the direct subscription path. Hence, our technique not only avoids misdetection as the K-value is always large enough. The K of the migrated ED is also only 17% too large at the beginning (70ms instead of the final 60 ms) and melts down quickly. For this example migration, it took 30s for all events to be detected at least once. Whereas a classic migration has to forward a total of 13,337 packets just for the single ED, our migration only forwards 51 packets and saves 99.6% of the network bandwidth and shipping overhead.

Hence, our method only forwards a minimal amount of events, and the old ED can im-



(a) The two alternating phases.

(b) Timeline of the network load.

Figure 6: Runtime optimization algorithm with two alternating phases.

mediately stop processing, as the new host takes over. Overload situations can always be mitigated by migration of event detectors to underloaded hosts.

5.3 Runtime Latency Optimization

To evaluate the runtime optimization we created a synthetic event processing hierarchy consisting of 8 EDs on 2 nodes, see Fig. 6(a). Arrows show event subscriptions, i.e., active event transmissions from one ED to another. The larger curved boxes depict the optimal allocation of EDs to nodes. For the synthetic benchmark after every 100s the EDs change their event subscriptions, i.e., the situation swings back and forth between the two sides of Fig. 6(a). For the two communication patterns different allocations of EDs to nodes are optimal.

Fig. 6(b) holds the resulting network load for 300 seconds. The upper straight line shows the network traffic for a fixed static ED distribution (0, 2, 5, and 7 on node #1, the others on node #2). The dotted line shows the network traffic of the ideal distribution.

Our runtime optimization (thin line) comes close. It finds the optimal ED distribution soon after the phases have switched and redistributes the EDs accordingly. Note that our runtime latency optimization approaches the optimum incrementally. The reason is that there is a cycle consisting of some measurements, 0.5s of cuckoo search, plus the actual ED migration. The cycle is repeated as the heuristic may not have found an optimum within the 0.5s or because the EDs' dependencies might have changed again in the meantime.

6 Conclusion

The presented building blocks of our EBS enable a low-latency processing of high data rate sensor streams to trigger actions with low delay. Our heuristics optimize the distribution of EDs over the available system resources at runtime. Packet loss has no influences on our algorithm but only on the event detectors. We also showed that our methods work well on sensor data streams from Realtime Locating Systems (RTLS) in a soccer application.

Future work will incorporate the ability to speculatively process a portion of events that would normally be withheld for a while due to the strict K-Slack approach. First results are promising and will further reduce detection latency.

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.

References

[AAB+05] D. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the Borealis stream processing engine. In *Proc. 2nd Conf. Innovative Data Systems Research*, pages 277–289, Asilomar, CA, 2005.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21th ACM Symp. Principles Database Systems*, pages 1–16, Madison, WI, 2002.

[CBB+03] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. etintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. 1st Conf. Innovative Data Systems Research*, Asilomar, CA, 2003.

[GFW+11] T. von der Grün, N. Franke, D. Wolf, N. Witt, and A. Eidloth. A real-time tracking system for football match and training analysis. In *Microelectronic Systems*, pages 199–212. Springer Berlin, 2011.

[GL12] A. Ghodrati and S. Lotfi. A hybrid CS/PSO algorithm for global optimization. In *Proc. 4th Asian Conf. Intelligent Inf. and Database Systems*, pages 89–98, Kaohsiung, Taiwan, 2012.

[HXZ07] J.-H. Hwang, Y. Xing, and S. Zdonik. A cooperative, self-configuring high-availability solution for stream processing. In *Proc. 23rd Intl. Conf. Data Engineering*, pages 176–185, Istanbul, Turkey, 2007.

[LLD+07] 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*, pages 67–74, Toronto, Canada, 2007.

[MP13a] C. Mutschler and M. Philippsen. Runtime migration of stateful event detectors with low-latency ordering constraints. In *Proc. 9th Intl. Workshop Sensor Networks and Systems for Pervasive Computing*, San Diego, CA, 2013.

[MP13b] Christopher Mutschler and Michael Philippsen. Distributed Low-Latency Out-of-Order Event Processing for High Data Rate Sensor Streams. In *27th Intl. Conf. Par. & Distrib. Processing Symp.*, Boston, MA, May 2013.

[PLS+06] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. 22nd Intl. Conf. Data Engineering*, pages 49–60, Atlanta, GA, 2006.

[TcZ07] N. Tatbul, U. Çetintemel, and S. Zdonik. Staying FIT: efficient load shedding techniques for distributed stream processing. In *Proc. 33rd Intl. Conf. Very Large Data Bases*, pages 159–170, Vienna, Austria, 2007.

[XZH05] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. 21st Intl. Conf. Data Eng.*, pages 791–802, Tokyo, Japan, 2005.

[YCJ06] Alex King Yeung Cheung and Hans-Arno Jacobsen. Dynamic load balancing in distributed content-based publish/subscribe. In *Proc. 7th Intl. Conf. Middleware*, pages 141–161, Melbourne, Australia, 2006.