DEBS Grand Challenge: The MSRG Soccer Monitoring Engine in the World Cup of Event Processing

Hans-Arno Jacobsen[1], Kianoosh Mokhtarian[1], Tilmann Rabl[1], Mohammad Sadoghi[2], Reza Sherafat Kazemzadeh[1], Young Yoon[1], Kaiwen Zhang[1]

[1]Middleware Systems Research Group (MSRG), University of Toronto
[2]IBM T.J. Watson Research Center

ABSTRACT

We present the design and implementation of a custom-built event processing system developed for the challenge of live event monitoring in a soccer match. We evaluated our system with a real workload and report on its performance. Our system can achieve a throughput of up to 790k input events per second, processing the game's sensor event stream up to 60 times faster than realtime. Furthermore, we investigated the applicability of several of the off-the-shelf general-purpose event processing engines to address the soccer monitoring problem. We additionally built two fully functional systems using Esper and Storm but concluded that the other engines are not expressive enough for our purposes.

1. INTRODUCTION

Complex Event Processing (CEP) systems in today's connected world define an exciting new area of research with rich potential applications and challenges. For the past two years, the ACM International Conference on Distributed Event-based Systems (DEBS) has organized a Grand Challenge competition targeted at promoting common ground and common evaluation criteria for CEP applications. The 2013 DEBS Grand Challenge [5], which is the focus of this paper, considers the problem of event monitoring in a soccer match. This is an application scenario that is reminiscent of a wide variety of use cases that are made feasible as availability and accuracy of small wireless sensors increase and the ability of CEP systems for live processing of sensor data improves. Furthermore, while our events of interest in a soccer match are domain-specific and related to detection of various game conditions, they are at the same time representative of the challenges in the wider spectrum of application scenarios that involve continuous stream processing.

In this context, this paper presents multiple approaches to address the 2013 DEBS Grand Challenge. Our solutions include an event processing system called *BlueBay* that we have designed and built for the analysis of soccer games. We discuss the modular design of BlueBay that enables easy plugging of various new types of soccer analysis queries. We report the performance of BlueBay in terms of throughput and delay, and its flexibility in tuning the trade-off between the two. This system can achieve a throughput of 350k events/sec with a 90-percentile per-event delay of only 0.005 ms, or alternatively a throughput of up to 790k events/sec if a higher delay is allowed (a 99-percentile delay of 15 ms).

In addition to the BlueBay engine, we considered several existing open-source off-the-shelf CEP engines and investigated their applicability to solve the Grand Challenge problem. We concluded that two of the existing CEP engines, namely Esper [2] and Storm [8], support queries that are sufficiently expressive for our soccer match monitoring purposes. We built additional systems (using Esper and Storm) and report on our experience in the development process.

In Section 2, we describe our multi-stage event processing pipeline that is intended to provide a unified framework under which our CEP solutions execute. In Section 3 we elaborate on different approaches that we considered in order to address the Grand Challenge monitoring problem. This includes discussions on Esper and Storm which resulted in working monitoring programs, as well as STREAM and StreamIT that we were unable to address the Challenge problem. Section 4 presents the main contribution of this paper, namely the BlueBay soccer analysis engine. Section 5 reports on our quantitative evaluation results of the BlueBay system.

2. MULTI-STAGE MONITORING PIPELINE

Figure 1 illustrates the overview of our soccer match monitoring pipeline consisting of three stages, namely, *(i)* the *sensor data collection and dispatching* stage, *(ii)* the *processing* stage, and *(iii) visualization and distribution* stage. We discuss each stage next.

2.1 Stage 1: Data Collection and Dispatching

The first stage in our monitoring pipeline involves sensor data collection and dispatching. The input sensor stream originates from transmitters attached to the ball, the referee's and players' feet, as well as the goal keepers' feet and arms. A sensor reading contains the sensor's unique identifier `sid`, its `x`, `y` and `z` space coordinate, its `vx`, `vy` and `vz` velocity vector components, its `ax`, `ay` and `az` acceleration vector components, velocity vector magnitude `|v|`, and acceleration vector magnitude `|a|`. Each sensor's event stream has a frequency of 200 events per second for the feet and arms transmitters, and 2000 events per second for the ball transmitter.

As shown in Figure 1, sensor readings collected from the soccer field can be fed into our monitoring pipeline either directly or indirectly. A direct feed is suitable for online monitoring of a live game. Alternatively, sensor data streams can be timestamped and logged into a file (currently in CSV format) and be fed into the pipeline at a later time. To support the offline processing mode, we developed a *network data dispatcher* program that reads the content of a game's sensor data log file and dispatches sensor readings over a socket connection.

2.2 Stage 2: continuous Query Processing

DEBS '13, June 29–July 3, 2013, Texas, USA.

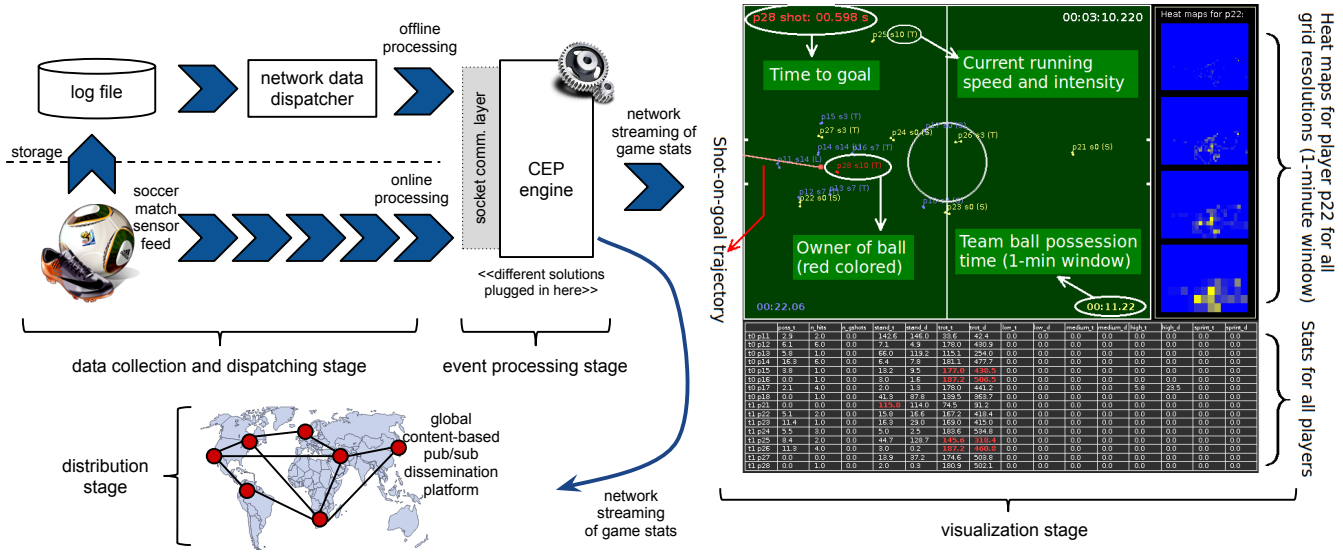

Figure 1: Unified soccer match monitoring pipeline.

The 2013 DEBS Grand Challenge defines four continuous monitoring queries that detect different game conditions (*e.g.*, shots on goal) and gather various game statistics (*e.g.*, ball possession). These queries are executed in the second stage of our pipeline using different CEP engines that we developed for this purpose. In what follows, we give a brief overview of these monitoring queries (for a more elaborate description, see [5]). We defer the discussion of different CEP engines to execute these queries to Section 3.

Query 1 (Q1) – Players' Running Performance: Q1 concerns monitoring of the players' running activity during the game based on 6 intensity classes (*i.e.*, *stop*, *trot*, *low*, *medium*, *high* and *sprint*). Players transition between these classes according to the momentary value of their speed. Q1 produces two types of outputs: *(i)* The continuous intensity statistics output is produced at a maximum frequency of 50 Hz, and *(ii)* the aggregate intensity statistics output is produced based on four different time windows.

Query 2 (Q2) – Ball Possession Statistics: Q2 computes the time that the ball is in possession of a player or team. The criteria that must be satisfied for a player to possess the ball is to hit it such that his foot is within 1 m of the ball and as a result of the hit the ball's acceleration reaches at least 55 m/s^2. Q2 produces outputs in two forms: *(i)* The per-player ball possession output reports the number of ball hits and the length of time each player possessed the ball, and *(ii)* the per-team ball possession output reports the percentage of time that the ball was in possession of each team.

Query 3 (Q3) – Heat Map: Q3 produces statistics of the aggregate time that each player spends in different parts of the soccer field (*a.k.a.*, the *heat map*). For this purpose, four grid structures with different cell sizes are defined dividing the field from just about a hundred cells (in the least granular grid) to several thousand cells (in the most granular one). Q3 outputs the percentage of time that each player spends in each cell over different time windows.

Query 4 (Q4) – Shots on Goal: Q4 detects players' shots on goal and produces an output stream that identifies the shooter and tracks the ball's motion towards the targeted goal post. The necessary conditions that constitute a shot on goal require that the ball be hit by a player (similar to Q2), remain within the field and the extrapolation of the ball's trajectory indicates that it reaches within the goal post's coordinates no later than 1.5 seconds after it was hit by the player (provided that other players do not divert the ball).

2.3 Stage 3: Visualization and Distribution

The final stage in the pipeline is to provide the results of the query processing stage to the end-users. We envision two usage scenarios for these results. The first scenario involves the use of a graphical user interface to visualize the movement of the ball and players in realtime and to display various statistics in sync with the game. We developed such a graphical monitoring panel that can be used by team coaches or TV reporters to analyze the game (see Section 2.3.1 for more detail). Alternatively, the CEP engine's output can be multicast over the Internet for soccer enthusiasts. We believe that a content-based pub/sub network [4, 6] is a suitable choice for this purpose. We now discuss these alternatives.

2.3.1 The Graphical Monitoring Panel

We have developed a GUI-based monitoring panel to visualize the soccer game and the related analysis information. In addition to a user friendly illustration of the information, this is a necessary tool for verifying the correctness of the different types of analysis (i.e., false positives and negatives). Given the huge volume of generated events, this verification is not feasible without proper visualization of the results, such as a detected ball hit, a running speed value, or a heat map. Figure 1 illustrates a screen shot of this tool, and a sample video is available at [9]. We have designed this interface such that it reflects the result of the important (and subtle) event processing steps, a few examples of which include:

- The detected running speed and intensity is shown next to each player. Besides verifying the speed values, this helped us find significant fluctuations in the detected intensities in small (20 ms) intervals if proper smoothing is not applied on raw data.
- Every time a ball hit by a player is detected, the ball turns red and gradually goes back to its original color (white). This reveals false positive and false negative detection of ball hits.
- The tool highlights the player identified as the ball possessor. Also, a timer tracks the reported ball possession time for each team (for the 1-minute window). This feature has contributed a lot to the high accuracy of our ball possession analyzer.
- Heat maps at different grid resolutions are displayed for a sample player (for the 1-minute window). Highlighted areas of the map track the player in the field, and his older positions gradually fade away as they fall outside the 1-minute window.

- Every time a shot on goal is detected, the ball's predicted trajectory and time to hit the goal is displayed on screen and continuously updated until we leave the shot-on-goal state. Moreover, before identifying as a shot on goal, for every ball hit, we visualize the estimated next-1-second trajectory which has helped in tuning and verifying the accuracy of our shot-on-goal detector.
- A table of statistics displays for each player the time and distance spent in each running intensity level, ball possession time and the number of ball hits. The table is continuously updated with recently changed values highlighted for easy tracking.
- The monitoring tool enables tracking of the missed sensor events, *e.g.*, no sensor data is received from player's feet for 500+ ms. This feature is necessary to identify that the root cause of some missed events (such as shots on goal) as incomplete sensor data—no foot sensor data suggests a close enough distance to the ball when it is shot towards the goal.

2.3.2 Pub/Sub-Based Dissemination Network

For the dissemination of query results, we envision the usage of a distributed content-based pub/sub system, such as PADRES [4] or Publiy [6]. Soccer fans around the globe can use the content-based filtering capability to tune into the statistics related to specific players or track various game conditions. For example, a subscription `sub=[player:'Ronaldo', condition:'ball-hits']` indicates the user's interest to track Ronaldo's ball hits. This allows the pub/sub network to filter out other statistics that are not of interest to the user. For this purpose, the output of the query execution stage (see Figure 1) is fed into the pub/sub network during the game. This stream is matched against subscriptions at each broker and flows towards the users according to their subscriptions.

```
insert into preprocessed_stream
select *,
  msrg.GameSetting.getId(s_id) as id,
  msrg.GameSetting.getType(s_id) as t_id,
  msrg.GameSetting.getSubtype(s_id) as subt_id
from msrg.EsperSensorEvent

insert into b_position_stream
select * from preprocessed_stream where t_id = 2

insert into b_relative_pos
select
  b.s_id as b_s_id, b.ts as b_ts, b.id as b_id,
  b.t_id as b_t_id, b.subt_id as b_subt_id,
  b.x as b_x, b.y as b_y, b.ax as b_ax, b.ay as b_ay,
  b.v as b_v, b.a as b_a, p.ts as p_ts, p.s_id as s_id,
  p.id as p_id, p.t_id as p_t_id, p.subt_id as p_subt_id,
  p.x as p_x, p.y as p_y, p.z as p_z,
  java.lang.Math.sqrt((p.x-b.x)*(p.x-b.x)+
    (p.y-b.y, 2)*(p.y-b.y, 2)) as dist
from preprocessed_stream as p unidirectional,
  b_position_stream.std:unique(s_id) as b
where (p.t_id = 3 or p.t_id = 4) and (b.t_id = 2)

create expression minDist
{(select min(dist) from b_relative_pos.std:unique(p_id))}

insert into b_possession
select *,
  msrg.GameSetting.getBallOwner(p_t_id,b_v,b_x,b_y) as owner
from b_relative_pos
where msrg.GameSetting.ballIn(b_x, b_y) = 1 and b_a > 0.5
  and minDist() = dist and dist <= 1000

insert into b_possession_percent
select *,
  sum(b_ts - prev(b_ts, 1)) as time_total,
  sum(b_ts - prev(b_ts, 1))
    * msrg.GameSetting.equalStr(owner,prev(owner,1),'teamA')
      as time_teamA,
  sum(b_ts - prev(b_ts, 1))
    * msrg.GameSetting.equalStr(owner,prev(owner,1),'teamB')
      as time_teamB
from b_possession.win:time(10 seconds)

select *,
  b_ts, owner, time_total,
  time_teamA/time_total as teamA_ownership_percent,
  time_teamB/time_total as teamB_ownership_percent
from pattern [every timer:interval(1 second)] unidirectional,
  b_possession_percent.std:unique(owner)
```

Figure 2: Challenge Query 2 written in Esper.

3. USE OF EXISTING CEP ENGINES

We now discuss the results of our investigation in the application of different off-the-shelf CEP engines to solve the Grand Challenge monitoring problems (see Section 2.2). We found that while some engines were sufficiently expressive to address our query processing needs, others were not suitable for our purposes.

3.1 Esper

We successfully used the Esper system [2] (version 4.9.0) to implement all four queries in the challenge. The Esper distribution comes with a complex event processing (CEP) engine and an event processing language (EPL) which provides a powerful and scriptive interface into the vast capabilities of the CEP engine. The approach we took in our implementation is based on logical decomposition of the each of the challenge query into smaller subqueries (formulated as EPL statements). The subqueries build on top of each other incrementally to produce the final output. At a high-level, this process involves the following phases:

Preprocessing: We first carry out a preprocessing phase in which the *raw* input sensor data stream is *augmented* with game information and tags that identify the object type (*i.e.*, ball, referee, foot or hand sensor), player id, and team association. These mappings are based on the game metadata information supplied in [5].

Processing: We decomposed each challenge query into several subqueries that are continuously evaluated on the augmented sensor stream. These subqueries incrementally compute the final outcome (Figure 2 illustrates formulation of Q2 in Esper's EPL).

Reporting and output: We defined time-triggered subqueries that produce a sampled output of each challenge query results based on the frequencies specified.

We observed that the Esper EPL features elaborate constructs (*e.g.*, windows, contextual partitioning, aggregation, expressions, etc) that are directly applicable towards our stream processing needs. Furthermore, the Esper framework supports seamless integration with the Java language in at two key levels. The first level facilitates interfacing with the CEP engine for the input and output of events to the engine. The second level allows incorporation of Java code snippets and function calls as part of the evaluation of an EPL query. We used the former capability to inject raw sensor readings from our network data dispatcher and collect the computed output results using a Java callback method. We used the latter capability to encode the game-specific domain knowledge as stateless Java static functions (*e.g.*, computing the ball trajectory). This approach was motivated by the higher readability of Java code.

Finally, we would like to reflect on our experience in working with Esper. Esper's high-level language gives great flexibility allowing for fast development and ease of change. In fact, after a sharp learning curve for us to become familiar with its capabilities, we were able to formulate all queries in a matter of a few days time. We believe that in a usecase with changing requirements an Esper-

`getId(s_id)`	Returns unique id for players, referee & ball wearing sensor `s_id`.
`getType(s_id)`	Returns type of entity (*i.e.*, team of players, referee & ball) wearing sensor `s_id`.
`getSubtype(s_id)`	Returns id for showing where sensor `s_id` is worn (*i.e.*, left/right leg/arm).
`equalStr(a, b, c)`	Returns `1` if `a=b=c` and `0` otherwise.
`ballIn(x, y)`	Returns `1` if `x, y` is a coordinate within field boundaries.
`getBallOwner(t_id, v, x, y)`	Returns `'ballout'` if `(x, y)` is within field or `'stopped'` if `v!=0`. Otherwise, returns `'teamA'` or `'teamB'` based on `t_id`.

Figure 3: Java functions from `msrg.GameSetting` used in Figure 2.

Figure 4: Storm topology excerpt for Query 3

based solution can be of great value. Also, as an added advantage of Esper's ease of integration with the Java language our Esper-based solution can be incorporated seemlessly in other programs.

3.2 Storm

Storm is a distributed stream processing system built for real time web scale stream processing [8]. A Storm program (*topology*) is a directed acyclic graph that consist of data sources (*spout*) as roots and data processing nodes (*bolts*) as inner nodes and leaves. Spouts emit tuples in a specified format that are consumed and processed by bolts. Spouts and bolts can have multiple instances that run in parallel. The way data is streamed to these instances is specified by *stream groupings*. The most basic stream grouping is a *shuffle grouping* that randomly distributes data to instances of bolts. A more advance grouping is the *field grouping* that send tuples with equal values in a certain field always to the same bolt. A Storm topology can be specified in various programming languages, for a more detailed discussion see [7].

Although spouts and bolts have to be individually implemented, Storm takes care of the distribution of tasks (spout or bolt instances) and the reliable transmission of tuples between the two. A Storm system consists of a master node that takes care of distributing tasks and code and worker nodes that execute subsets of a topology. A Storm cluster is backed by a Zookeeper cluster [3], which keeps all state information and makes the cluster reliable and fail-fast.

We implemented all 4 queries in Storm using Java. Figure 4 shows part of the topology that computes Q3 (heat maps). Since all data comes from the sensor stream there is only a single spout. As a first step, the sensor readings have to be matched with the players, this can be done with an arbitrary degree of parallelism using shuffle grouping. Next, the sensor readings from the two feet of each player have to be joined to a single position, this is done with a field grouping and a maximum degree of parallelism of the number of players. From the stream of each player's position the individual heat maps are computed and sent to the output bolt which sends the results to the GUI or standard output.

3.3 StreamIT

StreamIT [10] provides a high-level language with stream manipulation primitives. We unsuccessfully attempted to implement the challenge queries using StreamIT. We now list the limitations of StreamIT which prevented us from completing the queries:

Limited I/O support: StreamIT has restricted I/O capabilities.

Lack of powerful stateful operators: Communication between blocks in a StreamIT pipeline is regulated through different queues. Therefore, each block is essentially stateless and accesses additional data by peeking at various queues. This queue-based approach is not suitable to formulate the finite-state machine model required for our monitoring queries.

Lack of library support: We believe that it would have been more beneficial if StreamIT was provided as a library, where the code produced by StreamIT could be modified in its lower level form.

3.4 STREAM

STREAM is a data stream management system to evaluate continuous query developed by Stanford [1]. The main features of STREAM are to support a declarative continuous query language (CQL) that unifies access to incoming (structured) data stream and traditional stored data (in form of tables).

The CQL language is formulated over either streams (an unbounded bag of events) or relations (a finite time-varying bag of events). The CQL language is also extended with a sliding time- and count-based window semantics, essentially, the sliding window is a snapshot of an observed finite portion of the event stream. There are three classes of operators in CQL language in order to unify seamlessly access both streams and relations. These classes of operators are distinguished based on their input/output semantics: (1) relation-to-relation, which takes relations as input and produces a relation as output; (2) stream-to-relation, which takes streams as input and produces a relation as output; and (3) relation-to-stream, which takes relations as input and produces a stream as output. Another notable feature of CQL is the partitioning operator that partition a stream into a set of substreams based on the values of selected attributes in the event stream. Despite the novel features of STREAM, the lack of direct support for user-defined functions (needed to implement complex Finite-state machine behaviour required by DEBS Challenge queries) prevent us from including STREAM in our evaluation.

4. THE BLUEBAY ENGINE

BlueBay is the event processing engine that we have developed for analyzing soccer game sensor data. This engine is designed modular to serve as a general framework for adding any new type of soccer analysis query fairly easily (some discussed below), while it is also optimized for speed: With an efficient implementation in C++ it is now achieving a throughput of up to 790k events per second, handling the sensor events of up to a whole minute in only one second, i.e., 60 times faster than real time.

BlueBay Components. A number of the components of the BlueBay system are briefly described in the following. An *Active-Ball Tracker* tracks which of the ball sensors is the one we should be monitoring; only one of the multiple sensor equipped balls is the one being played with. There are few-second periods where there are temporarily more than one ball in the field, during which the Active-Ball Tracker should not be misguided until the extra ball is sent out. Once identified which ball sensor is relevant, a *Ball Tracker* tracks the state of the ball—position, speed and acceleration, denoised with proper filters (exp-weighted moving average). Since there is only one ball sensor that is on the surface of the ball, the position and velocity of the ball are very sensitive to rotations of the ball. This results in inaccurate trajectory predictions which we have alleviated using these filters. The Ball Tracker also monitors the acceleration of the ball and detects potential ball hits. We have found that the acceleration is highly sensitive to when ball hits the ground, hence we take only the XY-plane acceleration into account (based on a threshold of 75 m/s^2). BlueBay also includes a *Player Tracker* per each player which tracks the position of the foot sensors and the speed of the player (averaged over the two feet). The speed is denoised with a sliding window moving average filter to cut out the significant fluctuations in the speed value as the player takes every step. A new {running speed, duration} value is output every time the speed value is steady enough, with an output frequency bounded with a maximum of 50 Hz and a minimum of at least 5 Hz. Upon every potential ball hit, the hitting player is identified based on foot sensor positions. Then, a *Trajectory Estimator* predicts the trajectory of the ball in the next few seconds. The Trajectory Estimator takes into account the current (denoised) position and speed of the ball as well as gravity and air resistance.

Stream Window. A common data structure used in the different components is a sliding window which keeps timestamped data values—entries of the form {timestamp, value, duration}—such as a 1-minute window on how long a player spends in a grid or the

time a player is running at each intensity level. The key aspect of this window, to which we refer as *Stream Window*, is its efficiency: it performs all insertion and retrieval operations in constant time, and more importantly consumes only constant memory. This is achieved by bucketizing (discretizing) the timestamps at some granularity, and maintaining a list (in practice a circular buffer) of fixed length, irrespective of the input rate. For instance, we gather all events with a timestamp between 1000 ms and 1999 ms in one bucket, without losing any noticeable precision compared to the 5-minute length of the window; at the worst when we are pushing some outdated events out of the window they may be leaving up to 1 second late. The bucket length is chosen with respect to the total window length such that the required memory remains constant (e.g., a circular buffer of size always 100).

Query Analyzers. Given these and a number of other components, we can conduct different types of analysis fairly easily. Query 1 of the challenge can be readily responded using Stream Windows and the output of the aforementioned Player Tracker—a Stream Window for each defined running intensity level. Similarly for Query 2, we receive every ball hit with the corresponding hitter ID from the Ball Tracker, and we can track the per-player and per-team possession time with a simple counter and a Stream Window, respectively. The heat map analyzer (Query 3) is nothing but a collection of Stream Windows tracking data received from Player Trackers. These windows, of different lengths and for the different grid resolutions, a total of 34,000 windows, easily fit into a few hundred MB of memory and perform all operations in constant time. Finally, the shot-on-goal analyzer (Query 4) only needs to wait for ball hits from the Ball Tracker, and compare the trajectory given by the Ball Tracker with the position of the goals. This analyzer consists of a finite state machine. Every time the ball is hit, we enter a waiting state for up to a maximum distance (50 cm) or a maximum time (1 second) since the hit. At that point, we can have a reasonable estimate of the ball's trajectory and time to hit the goal, according to which we either enter the shot-on-goal state or go back to the no-shot state. In the former case, the trajectory is monitored upon every ball sensor event until the ball changes direction, gets too slow or leaves the field.

More Types of Analysis. The components introduced above enable various new types of soccer analysis, such as the following.

- A player's running statistics with and without the ball, and at the time of attack and defense can be analyzed using a counter/Stream Window on the Player Tracker's output.
- Ball passes and their success rate for each player and each team can be counted: We are signaled when a new player receives the ball (upon his first hit), and we know who has hit the ball last.
- Offsides can be detected by waiting for ball hits and the hitter ID from the Ball Tracker, and form a black list of players who are not allowed to receive that pass based on their positions at the time of hit (from the Player Trackers).
- The success of a player for man-marking his assigned opponent player can be analyzed through a Stream Windows to which we periodically push the distance of the two players. These windows tell us, for example, that defender X was within a 2-meter distance of the assigned attacker 90% of the time.
- The defense-to-attack time for a team to reach the opponent's 1/3rd of the field can be tracked by monitoring the Ball and Player Trackers and maintaining a single time counter and a flag for when, which team last acquired the ball (similar to how Query 2 analyzer monitors the Ball and Player Trackers). An event is emitted once the ball enters the opponent's defense area.

Efficiency. Since almost all of the operations in this event processing challenge can be done in $O(1)$ time, it is a matter of efficient implementation techniques that enables a throughput boost in the processing of such volume of events. A few of these techniques employed in the BlueBay system are as follows. First, the different components need to track per-player and per-sensor data. We avoid any hash map lookup (even though it is of $O(1)$) by assigning an *index* $\in [0, \textit{Num Entities} - 1]$ to each entity, such as a sensor or a player. The index for each sensor/player ID is looked up only once per event in an ID-to-index hash map. Then, all the remaining entity lookups (e.g., updating the position of a player) is done using array-based maps. Moreover, we note that the main body of the analysis consists of a substantial amount of mathematical operations. We avoid the unnecessary use of floating point operation where int64 precision would just suffice, while carefully handling cases where there is a risk of overflow (e.g., multiplying two picosecond values). We use efficient array-based circular buffers instead of linked lists or elastic vectors wherever the data size is fixed (which is often the case—Stream Windows). Furthermore, we run a separate process for reading data from the file, which simulates the source of sensor events in a live scenario. This process is run a few minutes before the BlueBay engine, and reads ahead the data from file in full or part (with a user-specified memory budget) to minimize disk I/O in the critical path of processing events.

Parallelization. The BlueBay engine can run in both single and multi-threaded mode. For the latter, we note that the events emitted to different output streams (per query and sometimes per sub-query) should not emit *reordered timestamps*. Thus, we do not run an arbitrary-size thread pool to handle the events. Rather, we run one thread per query, with the ball possession and shot-on-goal analyzer combined in one since they include common steps. Our code profiling analysis has shown that the most time consuming event processing step in BlueBay is the *emission of heat map statistics*—over half a million Stream Windows that should emit data. Even just iterating over all the windows for all players takes up to a few dozen milliseconds. We therefore designate a number of sub-threads in the thread handling the heat map query, which are responsible for the different grid resolutions. Reordering of output events across different grid resolutions is fine since they belong to different output streams. Once the emission timer fires, the heat map thread launches these sub-threads and blocks until they are all done, before handling any new heat map input event. We also note that in all query analyzers, we disable emission to stdout or file by default, since it involves significant IO that will not let us analyze the actual performance of the event processing engine. Note that, however, we do perform all the string formatting and preparing the final output; we just do not print it (i.e., `sprintf` to memory buffers rather than `printf` to stdout).

5. EVALUATION

We analyze the performance of the BlueBay system in terms of throughput and per-event delay—the time it takes from when we receive an event from the Data Dispatcher to when we have finished processing it and possibly generated an output event. Our evaluations are conducted on a PC with Intel Xeon 3.20 GHz 4-core CPU with 6 GB of memory. We measure the throughput as the number of events processed per seconds (e/s), and as a *speedup*. A speedup of s indicates that sensor events of s seconds (w.r.t. sensor timestamps) are processed in 1 second of actual time, i.e., events processed s times faster than real-time.

The BlueBay system in the non-threaded mode achieves an average throughput of 364k e/s, and an average speedup of 27.6x. The instantaneous throughput is shown in Figure 5. By enabling the multi-threaded mode, the performance is on one hand increased and on the other hand impacted by some noticeable overheads as

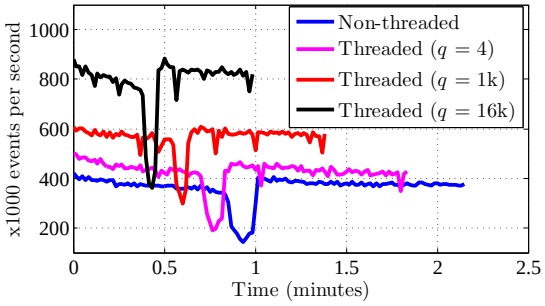

Figure 5: Throughput of the system (best viewed in color). The X axis represents actual time (hence higher-throughput cases ending earlier). The drop in the middle corresponds to the few minutes of missing ball sensor events, whereas the processing-heavy events (emitting heat map stats) are still present.

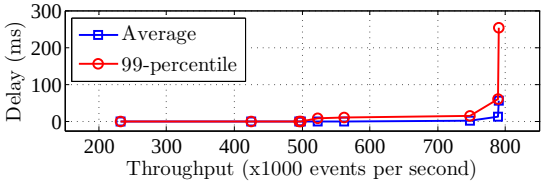

Figure 6: Tradeoff between throughput and per-event delay. Points on the graph represent experiments with different thread queue sizes: 1, 4, 16, 256, 1k, 4k, 16k and 64k.

discussed shortly. The overall throughput can be increase to 2+ times higher—over 790k e/s.

As described earlier, in the multi-threaded mode we allow a query of type X be processed while a query of type Y is taking some time. We can enforce the different query threads to run at more or less the same pace by limiting the input queue size of the threads. A queue size of 1 enforces that no thread can start working on a new event (it is not given one) until another thread finishes the previous event. A queue size of ∞ makes the threads completely independent. The queue size also governs a tradeoff between throughput and per-event delay: A limited queue size ensures that when a thread is falling behind (typically the event triggering heat map emissions), the other threads are paused after some point, giving the full CPU to the busy thread (sub-threads handling heat map emissions; see Section 4). In Figure 5 we see the performance of the threaded BlueBay with different queue sizes (q).

Note that unlike IO-involved jobs where threading immediately boosts the performance by avoiding the waste of any CPU cycles, in here all threads carry out CPU-heavy jobs, so for small queue sizes the overhead of threading is more significant than the obtained throughput increase in Figure 5. A major part of this overhead is that of the safe enquing/dequeing of events for the threads which takes a non-negligible time compared to the only-a-few-microsecond processing time for most events. Another, less significant factor is the slight performance drop by turning on additional monitoring features such as collecting delay information and tracking the 99-percentile delay; turning on these features in the non-threaded mode drops the aforementioned throughput of 364k to 346k e/s. Finally, unlike IO-involved jobs, in here the overhead of the many context switches between the CPU-intensive threads is non-negligible.

The tradeoff discussed above between throughput and delay is illustrated in Figure 6. We can observe in this figure that the throughput can be increased only up to some point, after which some threads (the heat map thread) has a hard time catching up; hence the jump in the delay value. In particular, the average/99-percentile processing delay for last three queue sizes, 4k, 16k and 64k is 2.9/15 ms, 13/61 ms and 56/253 ms, respectively. The average throughput for these cases is 747k, 789k and 790k e/s. This pattern which can be seen in Figure 6 shows the maximum throughput. For the non-threaded version (not shown in the figure), the average/90-percentile delay and throughput are 0.004/0.005 ms and 346k e/s, respectively.

It is noteworthy that the rather complex relationship between throughput, delay and queue size is mainly due to: (i) the substantial unevenness between the work taken for the different events—in particular the *emission* of heat maps (even only preparing the output without the final push to stdout/file), and to a lesser extent, the running statistics—and (ii) the enforcement of one thread per query (and one per sub-query for heat maps) to avoid reordering of timestamps in the output streams; described in Section 4.

Summary. Among the different scenarios described, we recommend the use of BlueBay in non-threaded mode for analyzing live soccer games, as it can already process **27+ times faster** than real time with less than **a few microseconds of delay** for 99% of events (75 ms max delay which is for heat map emission). On the other hand, for offline analysis of pre-recorded events where a slight delay for some events is not a problem, the multi-threaded mode with a queue size of **4k** (or **16k**) is suggested, which provides a throughput of **747k e/s** (/**789k e/s**), a speedup of **57x** (/**60x**) and a 99-percentile delay of **15 ms** (/**61 ms**).

6. CONCLUSION

We have presented the BlueBay event processing system for the analysis of soccer games. This system serves as a framework for conducting various types of analysis on sensor data received from a soccer match, including the queries specified in the Grand Challenge and additional, new queries. BlueBay provides flexibility in trading off between the event processing throughput and the per-event processing latency. Our implementation on a commodity PC can achieve a throughput of up to nearly 800k events/sec. In addition, we investigated several existing CEP engines and their applicability to the Grand Challenge problem. We found that Esper [2] and Storm [8] support queries that are expressive enough for our soccer analysis purposes. We built additional systems (using Esper and Storm) and reported on our experience in the development process. An extended version of this paper including all performance results and system setups will be published as a technical report.

7. REFERENCES

[1] A. Arasu et al. Stream: the stanford stream data manager (demonstration description). In *SIGMOD'03*.
[2] Esper Tech Inc. The Esper complex event processing platform.
[3] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC'10*.
[4] H.-A. Jacobsen, A. K. Y. Cheung, G. Li, B. Maniymaran, V. Muthusamy, and R. S. Kazemzadeh. The PADRES publish/subscribe system. In *Principles and Applications of Distributed Event-Based Systems*. IGI Global'10.
[5] Z. Jerzak and K. Kadner. DEBS 2012 grand challenge.
[6] R. S. Kazemzadeh and H.-A. Jacobsen. Opportunistic multipath forwarding in content-based publish/subscribe overlays. In *Middleware*, 2012.
[7] J. Leibiusky, G. Eisbruch, and D. Simonassi. *Getting Started with Storm*. O'Reilly Media'12.
[8] N. Marz. Storm - distributed and fault-tolerant realtime computation. `http://www.storm-project.net/`.
[9] Middleware Systems Research Group. Soccer game monitoring – sample video, 2013. `http://msrg.org/datasets/blue-bay`.
[10] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*.