DEBS Grand Challenge: Glasgow Automata Illustrated

Alexandros Koliousis
School of Computing Science
University of Glasgow
alexandros.koliousis@glasgow.ac.uk

Joseph Sventek
School of Computing Science
University of Glasgow
joseph.sventek@glasgow.ac.uk

ABSTRACT

The challenge is solved using Glasgow automata, concise complex event processing engines executable in the context of a topic-based publish/subscribe cache of event streams and relations. The imperative programming style of the Glasgow Automaton Programming Language (GAPL) enables multiple, efficient realisations of the two challenge queries.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*

Keywords

Complex event processing, measurement

1. INTRODUCTION

The DEBS '12 Grand Challenge (DGC) organisers invited event-based systems to engage in a competition by solving two continuous queries with regards to monitoring large, high-tech manufacturing equipment. We took up the challenge using our topic-based publish/subscribe system that was originally developed for the purposes of home network management [1]. Nonetheless, this challenge was a timely opportunity to apply our system and its complex event pattern programming language, GAPL, to a different application domain.

A program in GAPL, termed an *automaton*, is not a mere collection of predefined, nested, complex query operators. The language rather exposes a number of memory and control structures that enable programmers to construct them on the fly. This imperative programming style of Glasgow automata is also the highlight of our solution, as it enables aggregations of the proposed twenty-two query operators of the challenge.

The paper provides a brief tour of the cache and its automaton programming language in §2. Once raw events (measurements from manufacturing equipment) enter the cache, a process described in §3, the paper describes how the output streams of the two queries are derived in §4. The latency and throughput of the proposed DGC solution is evaluated in §5. Finally, the paper concludes in §6.

DEBS '12, July 16–20, 2012, Berlin, Germany.

2. THE CACHE IN A NUTSHELL

The keystone of our system is a topic-based, pub/sub cache, written in C. Topics are organised in memory as tables, representing either append-only streams or static relations, each of them associated with a schema. E.g.,

```
1 # A stream 'S' and a relation 'R'
  create table S (i integer, b boolean, r real)
3 create persistenttable R (k varchar primary key, v...)
```

The ordering of topical events is the time of insertion. The rows of a stream are ephemeral, stored in a circular buffer and overwritten whenever the buffer wraps. The rows of a relation, on the other hand, are persistent, stored in the heap and overwritten explicitly on duplicate key:

```
4 insert into S values ('1', 'true', '0.1')
5 insert into R values (''A'', '1') on duplicate key update
```

At any point in time, a user process can select past events from the cache or subscribe an interest in complex patterns of future events. The ability to submit ad hoc `select` statements, appropriately enhanced with time windows, enables programmers to periodically harvest events from the cache for storage or display purposes [1]. The thrust of our DGC solution, however, exploits the other aspect of the system – *Glasgow automata*.

Automata are programs written in GAPL, where users specify complex patterns over the cached streams and relations. Programs compile into instructions for a stack machine; if compilation is successful, each compiled automaton is bound to a separate execution thread.

Streams and relations are globally accessible in automata by means of subscriptions and associations, respectively. Local state is stored in variables:

```
6 subscribe e to S;
  associate r with R;
  int x;
  bool b;
  initialization { x = 0; }
  behavior {
    x = e.i;
    b = hasEntry(r, Identifier('A')); # true if R.k = 'A'
14 }
```

The `behavior` clause of an automaton is executed each time an event is inserted into a topic of interest. Therefore, an automaton must subscribe to at least one topic. The `initialization` clause is executed once, when the execution thread

is created, usually to initialise any local variables. As events arrive, their attributes are accessible using the dotted notation, with names taken from the corresponding schema.

The language is strongly typed. For example, an assignment `x = e.r` in the example above would cause an execution error (since attribute `r` of `S` is a real) and the automaton thread would terminate. Types are either basic (e.g. an `int`, a `string`) or aggregates (e.g. a `window` of integers, a `sequence` of strings).

An automaton can populate the cached streams and relations via the `publish` and `insert` procedures, respectively:

```
15 behavior {
     publish('S', 2, false, 0.2);
     insert(r, Identifier('B'), value);
18 }
```

Besides storing derived events in the cache, an automaton can also transmit them to its registering process. A subscriber always registers details of a callback service (viz., a host address, a port, and a service identifier) together with the automaton source. This way, an automaton can publish events back to the subscriber using the `send` procedure:

```
19 subscribe e to S;
20 behavior { send(e); }
```

Two recurring aggregate types in our solution are `sequence` and `window`. A sequence is an unbounded, ordered list of arbitrary basic data type instances. The elements of a sequence are accessible via the `seqElement` function, starting at index 0. The `append` function adds elements to a sequence:

```
21 sequence s;
   int x;
   initialization {
     s = Sequence(1, true);  # int & bool
     x = seqElement(s, 0);   # x = 1
     append(s, 0.1);         # seqElement(s, 2) = 0.1
27 }
```

A window, on the other hand, is an ordered list of instances of a particular data type, constrained either to a fixed number of items or a fixed time interval. A window is also populated with the `append` function, but the window "slides to the right" when filled to capacity. The elements of a window are accessible via an iterator:

```
28 window w;
   iterator i;
   real x;
   initialization {
     w = Window(real, ROWS, 1); # a window of a real
     append(w, 0.1);
     append(w, 0.2); # the window slides; 0.1 is removed
     i = Iterator(w);
     while(hasNext(i)) {
       x = next(i);
     }  # x = 0.2
39   destroy(i); # not strictly necessary
40 }
```

Certain variables are instantiated with constructors, like the `Iterator` above. Although `destroy` is available to deallocate them, the run-time system also automatically garbage collects any dynamically allocated memory.

This concludes our quick overview of Glasgow automata. The following sections present code listings that will shed further light into the aforementioned features. Unfortunately, we found no use for associations, the cache's global key-value stores, or for the aggregate type `map`, an automaton's local key-value store, in solving the DGC queries. These are powerful constructs for expressing stream-to-relation operators. The challenge, however, relies mostly on stream-to-stream operators. GAPL also supports a wealth of built-in functions, only some of which are used here.

3. DATA POPULATION

The cache is populated with measurements from manufacturing equipment stored in a text file – one measurement per line – and ordered by their time of occurrence. For the purposes of the challenge, the measurement data reach the cache via two proxy programs.

The data collection was accompanied by two Java programs, a data generator and a data receiver. The data generator reads the text file and replays measurements in a way that emulates their real-time rate of events (~ 100 events/s). The generator encodes the data as Google Protocol Buffers[1] and transmits them over a TCP socket. The organisers also provided a simple server that receives these protocol buffers and decodes them into Java objects.

The cache receives events from population applications via RPC channels. The supplied data receiver was extended to open an RPC channel to the cache and translate measurements from the Google interchange format to `insert` statements, the format understood by our system, generating one RPC/measurement. At the real-time rate of events, the overhead of our RPC mechanism on the modified receiver is negligible, achieving a maximum rate of ~ 7000 events/s. The schema for measurements in the cache is shown below. Only those attributes required by the two DGC queries are inserted into the cache.[2]

```
41 create table CDataPoint (time tstamp,
     mf01 integer, mf02 integer, mf03 integer,
     bm05 integer, bm06 integer, bm07 integer,
     bm08 integer, bm09 integer, bm10 integer,
     pp04 integer, pp05 integer, pp06 integer)
```

4. QUERIES

The DGC organisers have sketched two query plans using a set of operators that generate and consume a number of intermediate event streams, derived from the initial measurement stream, before finally producing the output streams. The imperative programming style of GAPL enables multiple realisations of these query plans, e.g. by merging two or more operators together or by omitting certain intermediate event streams; other combinations of streams and operators are discussed later, in §4.3. For now, the automata presented herein remain true to the two sketches, generating intermediate event streams as requested.

The challenge specified twenty-two operators, generating twelve temporary streams. Figure 1 is a representation of the aforementioned query diagrams where circles represent operators and squares the events they generate. The schemas for those events precede §4.1. It is assumed that the final output streams are sent back to the process(es) that have registered the corresponding automata. For economy of space, the implementation of repetitive operators is omitted from the discussion.

[1] `http://code.google.com/p/protobuf`

[2] For reasons of simplicity, the boolean attributes `bm05`, ..., `bm10` are encoded as integers, with `-1` representing their initial unknown value (cf. $l.50$).

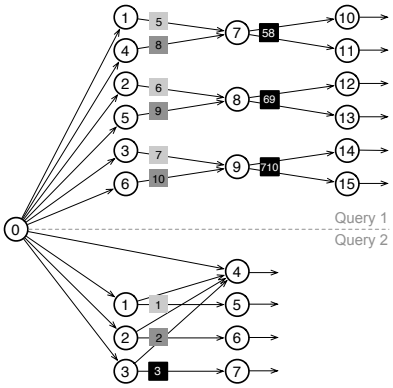

Figure 1: Queries illustrated.

```
42 create table S5 (time tstamp, edge integer)
   # S6-S10 are identical to S5
   create table S58 (time tstamp, dt integer)
   # S69 and S710 are identical to S58
   create table S1 (time tstamp, ave real, rng real)
47 # S2 and S3 are identical to S1
```

4.1 Query 1

The first query consists of fifteen operators. Its first six operators use the six attributes `bm05`, ..., `bm10`, respectively. The automaton that implements **operator 1** follows. Its function is to detect a value change – from `false` (0) to `true` (1), and vice versa – in the sensor reading `bm05`. Upon a change, the operator emits an event (a rising or falling edge), along with its time of occurrence, to stream `S5`.

```
48 subscribe e to CDataPoint;
   int bm05;
50 initialization { bm05 = -1 } # unknown previous value
   behavior {
      if (bm05 == 0 && e.bm05 == 1) {
         publish('S5', e.time, 1); # rising edge
      } else if (bm05 == 1 && e.bm05 == 0) {
         publish('S5', e.time, 0); # falling edge
      }
      bm05 = e.bm05; # store variable
58 }
```

Two variables suffice to detect a value change of attribute `bm05`. Variable `e.bm05` holds the current value of the attribute in question; and variable `bm05` holds the previous one. When a change is detected, the automaton publishes an event to the intermediate event stream `S5`. The `publish` procedure calls presuppose the creation of table `S5` in the cache (cf. *l*.42). **Operators 2-6** have identical functionality and are omitted.

The temporary streams `S5` and `S8` generated by operators 1 and 4 are consumed by **operator 7** (Figure 1). In the following automaton, the state change in sensor `bm05` is correlated with that of the valve `bm08`, calculating the time difference between the two events.

```
59 subscribe s5 to S5;
   subscribe s8 to S8;
   int e5, dt;
   tstamp t5;
   initialization { e5 = -1; }
   behavior {
      if (currentTopic() == 'S8') {
66       if (e5 == s8.edge) {
            dt = tstampDiff(s8.time, t5);
            publish('S58', s8.time, dt);
         }
      } else if (currentTopic() == 'S5') {
         e5 = s5.edge;
         t5 = s5.time;
      }
74 }
```

In GAPL, the union of the two or more streams is simply a matter of multiple subscriptions. When an automaton subscribes to two or more topics, the behavior clause differentiates between them using the `currentTopic()` function. Given the temporal ordering of events, it is straight-forward to write the pattern "event `s5` followed by event `s8`," conditional on the `if` clause in *l*.66 being true. **Operator 8** correlates streams `S6` and `S9` and **operator 9** correlates streams `S7` and `S10`, with similar implementations to operator 7. The remaining operators produce the output streams of Query 1.

Operator 10 and **operator 11** are considered in unison since both use the same local state, a 24-hour sliding window of `s58` events generated by operator 7. Within this time window, operator 10 raises an alarm whenever `s58.dt` increases more that 1%; and operator 11 monitors the line $y = a \cdot$ `s58.dt` $+ b$ using a least-squares linear regression. We first construct and populate this window. Recall that when the `append` procedure is called, the window slides to the right – here, those events that occurred 24 hours before `s58.time` will be removed.

```
75 subscribe s58 to S58;
   window w;
   initialization { w = Window(real, SECS, 86400); } # 24h 86400s
   behavior {
      append(w, float(s58.dt), s58.time);
80 }
```

The GAPL implementation of operators 10 and 11 depends on the possible number of elements in the window. One option, for example, is to compute the linear regression line using a window of sequences:[3]

```
81 n = 0;
   i = Iterator(windowofsequences);
   while(hasNext(i)) {
      s = next(i); # s is Sequence(s58.time, s58.dt)
      if (n == 0)
         first = seqElement(s, 0);
      x = float(tstampDiff(seqElement(s, 0), first));
      y = float(seqElement(s, 1));
      X = X + x;       # sum of x(i)
      Y = Y + y;       # sum of y(i)
      XY= XY+ (x*y);   # sum of x(i) y(i)
      X2= X2+ (x*x);   # sum of x(i) squared
      n += 1;
   }
   N = float(n);
   a = ( (N*XY) - (X* Y) ) / ( (N*X2) - (X*X) );
97 b = ( (Y*X2) - (X*XY) ) / ( (N*X2) - (X*X) );
```

Another alternative is to introduce a function, i.e. `lsqrf`, into GAPL's dictionary of built-ins that computes the coefficients a and b with a C implementation similar to the automaton above. E.g., given the window `w` of reals,

```
98 line = lsqrf(w); # line is Sequence(a, b);
```

[3]Rather than raising timestamps – unsigned 64-bit integers representing the nanoseconds lapsed since 1970 – to the power of 2, the automaton shifts the x-axis to the left, starting at $x = 0$. Given `first`, the first timestamp in the window, the computed linear regression line is $y =$ `a` $\cdot (x -$ `first`$) +$ `b`.

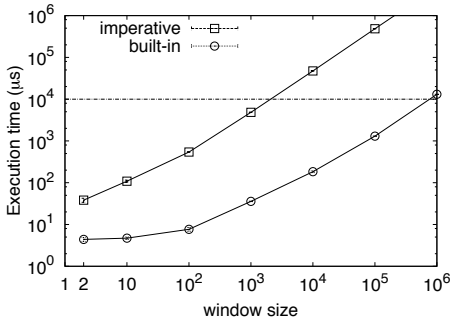

Figure 2: The imperative *vs.* built-in execution time of the least-squares regression line as a function of the window size on a 2.4GHz Intel Core i7.

Figure 2 shows the execution time of the two implementations as a function of the window size. The number of (x, y) points in the 24-hour window affects the real-time performance of our DGC solution since, at 100 events/s, automata have $10ms$ to execute their behavior before another raw event arrives. As illustrated, the imperative computation of the regression line finishes in under $10ms$ when fitting fewer than 1000 points to a line; in the same time, the built-in function can fit approximately 3 orders of magnitude more points. The difference is due to the number of instructions in the automaton's stack machine. In the absence of an estimate of the number of transitions, we have opted for the built-in choice in our solution.

The following automaton computes a and b, as well as the relative increase in `s58.dt`.

```
99  subscribe s58 to S58;
    sequence s;
    real a, b, max, increase;
    behavior {
103    append(w, float(s58.dt), s58.time);
       if (winSize(w) > 1) {
          s = lsqrf(w); # otherwise, replace with ll.81-97
          a = seqElement(s, 0);
          b = seqElement(s, 1);
          max = winMax(w);
          increase = (float(s58.dt) - max) / max;
          if (increase > 0.01)
             send('Alarm!', a, b);
          else
             send('OK.', a, b); # to be plotted
114    }
115 }
```

Note that the output streams of operator 10 and 11 have been merged under the same schema to ease processing at the subscriber – e.g., a plotter process. These streams can be split by implementing the two operators in separate automata, a trivial exercise. **Operators 12-13** and **operators 14-15** are pairwise similar (see Figure 1). This concludes the solution to the first DGC query.

4.2 Query 2

The second query consists of seven operators. The first three operators use attributes `mf01-mf03`. The following automaton implements **operator 1**. It calculates the average value and relative variation of the sensor reading `mf01` over a period of 1 second and outputs the results every 1 second. **Operator 2** and **operator 3** are similar.

```
116 subscribe e to CDataPoint;
    subscribe t to Timer;
    tstamp last;
    int min, max, n, sum;
    real avg, rng;
    initialization {
       min = 1234567890; # a value larger than life
       max = -1;
       n = 0;
       sum = 0;
    }
    behavior {
128    if (currentTopic() == 'Timer') {
          if (n > 0) {
             avg = float(sum)/float(n);
             rng = float(max - min)/float(max);
             publish('S1', last, avg, rng);
          }
          sum = 0;
          n = 0;
136    } else { # current topic is CDataPoint
          last = e.ts;
          if (min > s.mf01) min = e.mf01;
          if (max < s.mf01) max = e.mf01;
          sum = sum + e.mf01;
          n += 1;
142    }
143 }
```

In GAPL, there is a special topic, `Timer`, that delivers an event to its subscribers every second. Since the challenge requires operator 1 to publish events every second to stream `S1`, the automaton above is driven by such a timer event. Since only summary statistics are required, the code uses variables `sum`, `min`, and `max` to derive them, rather than a 1-sec window.

Operator 5 computes the per minute power consumption of the equipment, as measured by sensor `mf01` and accumulated by operator 1. Its corresponding automaton is shown below. The code consumes stream `S1`, generating output every 60 seconds. **Operator 6** and **operator 7** are similar.

```
144 subscribe s1 to S1;
    subscribe t to Timer;
    tstamp last;
    real pwr, onethird;
    int ticks;
    initialization {
       pwr = 0.0; ticks = 0;
       onethird = 1.0/3.0;
    }
    behavior {
       if (currentTopic() == 'Timer') {
          ticks += 1;
          if (ticks == 60) {
             send(last, pwr);
             pwr = 0.0; ticks = 0;
          }
       } else {
          last = s1.t;
          pwr = pwr + 208.0/power(s1.ave, onethird);
       }
164 }
```

Finally, **operator 4** records the sensor readings `mf01-mf03`, and when the relative variation (as computed by operators 1-3) of any one of them exceeds 30%, the operator must output these records. The recording starts 20 seconds before, and ends 70 seconds after the threshold violation. When multiple violations occur, the operator must always capture raw data 20 seconds before the first and 70 seconds after the last violation. The following automaton concludes the solution to the second query of the challenge.

```
165 subscribe e to CDataPoint;
    subscribe s1 to S1;
    subscribe s2 to S2;
    subscribe s3 to S3;
```

```
window w;
bool critical;
sequence s;
tstamp last;
real avg1, rng1, ...;
initialization {
   w = Window(sequence, SECS, 20);
   critical = false;
   avg1 = 0.0; rng1 = 0.0; ...
}
behavior {
   if (currentTopic() == 'S1') {
      avg1 = s1.ave;
      rng1 = s1.rng;
183   if (s1.rng > 0.3) {
         last = tstampDelta(s1.t, 70, FALSE);
         if (! critical) {
            critical = true;
187         send(w);
         }
189   }
   } else if (currentTopic() == 'S2') {...} # same as S1
   } else if (currentTopic() == 'S3') {...} # same as S1
   } else { # current topic is CDataPoint
193   s = Sequence(e.time, e.mf01, e.mf02, e.mf03,
         avg1, rng2, ...); # are these necessary?!
      if (critical) {
         if (tstampDiff(e.ts, last) < 0) send(s);
         else {
            critical = false;
            append(w, s, e.time);
         }
      } else append(w, s, e.time);
202   }
203 }
```

4.3 Putting it all together

The sequential execution of an automaton's behavior enables groups of query operators to coexist in the context of one execution thread. This section demonstrates how to fold the intermediate streams of the challenge (ll.42-47) in local state variables to produce one automaton per query.

To begin with, let us put together the operators of Query 1. As discussed in §4.1, operator 7 asks for "events `S5`, followed by events `S8`." This means that if `e.bm05` rises (or falls) at the i^{th} execution of the `behavior` clause, `e.bm08` will rise (or fall) at some later iteration $j > i$. In this temporal order, operator 7 can be grouped with operators 1 and 4 as follows.

```
204 subscribe e to CDataPoint;
   int bm05, bm08;
   sequence s5;
   initialization{ bm05 = -1; bm08 = -1; }
   behavior {
      if (
      (bm08 == 0 && e.bm08 == 1 && seqElement(s5, 0) == 1)
         ||
      (bm08 == 1 && e.bm08 == 0 && seqElement(s5, 0) == 0)
      ) {
         dt = tstampDiff(e.ts, seqElement(s5, 1));
215      publish('S58', dt, e.ts);
      }
      bm08 = e.bm08;
      # Rather than publish to S5, store edge in sequence
      if (bm05 == 0 && e.bm05 == 1) {
         s5 = Sequence(1, e.ts);
      } else if (bm05 == 1 && e.bm05 == 0) {
         s5 = Sequence(0, e.ts);
      }
      bm05 = e.bm05;
225 }
```

It is also possible to avoid the `publish` call to stream `S58` by embedding operators 10 and 11 - i.e., in the automaton above, replace l.215 with lines ll.103-114. It becomes apparent that by repeating the above code segments for variables `bm01`, ..., `bm06`, the fifteen operators of the first query can be implemented in just one automaton.

Let us now revisit the second query of the challenge. The first three operators are ideally synchronised because `mf01-mf03` are attributes of the same event `e`. In fact, since their implementations are identical (cf. ll.116-143), it should be `s1.time` = `s2.time` = `s3.time` when these temporary events are published. It is trivial to extend ll.136-142 to compute the minimum, maximum, and sum of variables `e.mf02` and `e.mf03` as well. Similarly, when a `Timer` event arrives, the automaton can computes the average and range of all three variables. The last operator to embed is operator 4 (cf. ll.128-136):

```
226 if (currentTopic() == 'Timer') {
      if (n > 0) {
         avg1 = float(sum1)/float(n);
         rng1 = float(max1 - min1)/float(max1);
         pwr1 = pwr1 + 208.0/power(avg1, onethird);
         sum1 = 0;
         ...
         # repeat for mf02 and mf03
         ...
         if (rng1 > 0.3 || rng2 > 0.3 || rng3 > 0.3) {
            ...
            # enter critical window as in ll. 183-189
         }
      }
      ticks += 1;
      if (ticks == 60) {
         send(last, pwr1, pwr2, pwr3);
         pwr1 = 0.0;
         ...
      }
   } else {
      ...
      # handle CDataPoint events as in ll.193-202
249 }
```

This concludes the discussion about grouping the twenty-two operators of the challenge into two Glasgow automata, where it has been demonstrated how GAPL can realise multiple user-defined execution plans of the same query. Other possible combinations are beyond the scope of this paper.

5. EVALUATION

This section evaluates the performance of the nineteen automata of §4.1 and §4.2 and the two automata of §4.3. Their performance is evaluated with regards to two measures, queueing delay and execution (or wall-clock) time, that affect the throughput and latency of our proposed GC solutions. The queueing delay is the difference between the time an event `e` is inserted into the cache, `e.tstamp`, and the time the event arrives at an automaton. E.g.,

```
250 subscribe e to CDataPoint;
   int dt;
252 behavior { dt = tstampDiff(tstampNow(), e.tstamp); }
```

The execution time is the difference between the time an automaton's behavior starts and the time it finishes processing an event. Experiments are run on a 2.4GHz Intel Core i7 with 8GB of RAM, running Mac OS X Lion 10.7.3. The experimental setup is the one described in §3.

As more automata subscribe to a topic, the queueing delay of its events increase. For example, ten out of the twenty-two operators of the challenge await the arrival of `CDataPoint` events (cf. Figure 1). Using one automaton per operator, the cache notifies ten execution threads upon insertion. Figure 3 illustrates this effect for the first 100,000 events of the data

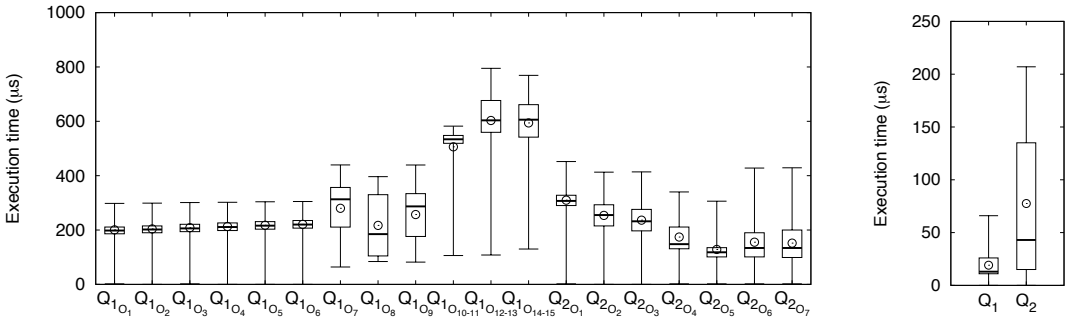

Figure 4: The minimum, 25^{th}, 50^{th}, 75^{th}, and 99^{th} percentile of the per automaton execution time.

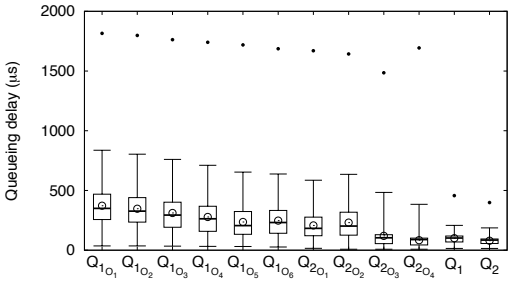

Figure 3: The minimum, 25^{th}, 50^{th}, 75^{th}, and 99^{th} percentile of the queueing delay of `CDataPoint` events per automaton. Points are average and maximum.

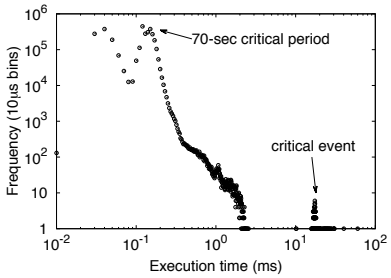

Figure 5: Q_2's frequency distribution.

collection. The speed of the data generator is 1, i.e. the insertion rate approximates the real-time rate of measurements (100 events/s). The `CDataPoint` events arrive first at operator 4 of Query 2 and last at operator 1 of Query 1. Although the delay when running nineteen automata is well within the $10ms$ threshold, using only two automata the mean delay decreases, as does its variance (cf. Q_1 and Q_2 in Figure 3).

In estimating the execution time of the automata, we speed up the simulation process by a factor of 100 and insert the first 5,000,000 events of the data collection. Figure 4 shows the per automaton execution time. It demonstrates that by grouping the twenty-two operators of the challenge into just two automata, Q_1 and Q_2, the execution time decreases. Fewer automaton threads run faster since `publish` or `send` procedure calls will cause an automaton to reschedule either the main thread or the RPC thread, respectively.

The main source of latency for Query 1 operators, the `lsqrf` function call, was discussed in §4.1. This discussion focuses on Query 2, and in particular the 1% of execution times for Q_2 not shown in Figure 4. It is important to characterise values greater than the 99^{th} percentile because some of them represent significant events.

Figure 5 shows Q_2's execution time frequency distribution. The most expensive operation of Query 2 is sending a 20-sec window of sequences when a violation is detected (l.187). At 100 events/s, the window contains 2000 events, each being the sequence:

```
253 Sequence(e.time, e.mf01, e.mf02, e.mf03, avg1...rng3);
```

The cost of sending a window of 2000 such events is $\sim 18ms$. Once this critical violation occurs, and for $70s$ afterwards, the automaton sends one such sequence at a time. The latter cost is $\sim 200\mu s$. These values are in agreement with those shown in Figure 5. Sending the window momentarily violates the $10ms$ threshold for the real-time system – but the system catches up in the next iterations. If, however, the last six reals are removed from the sequence, the cost of sending the window decreases below $10ms$. Being 1-second aggregates of the first three columns of the previous 100 events, these values appear redundant.

By speeding up the simulation process by a factor of 100, automata Q_1 and Q_2 process on average 5583 events/s. Thus, on this hardware, and given the true rate of 100 events/s, our implementation has the capacity to process $\times 55$ more events than it was originally intended to process in real-time.

6. CONCLUDING REMARKS

This paper has presented solutions for the DEBS 2012 Grand Challenge using Glasgow automata, user-defined programs for complex event processing, originally designed for home network management. In a potential deployment of our system at the manufacturing site, the system can easily cope with the real-time rate of measurements. The imperative automaton programming style enables different, user-defined optimisations of the query plan execution, some of which are more performant than others. This is to be contrasted with traditional complex event processing engines where the execution plan is determined by a fixed set of nested query operators.

7. REFERENCES

[1] J. Sventek et al. An information plane architecture supporting home network management. In *Proceedings of the IFIP/IEEE IM*, 2011.