Logical Optimisation and Cost Modelling of Stream-Processing Programs Written in a Purely-Functional Framework

Jonathan Dowland
Red Hat, Inc.
3 Science Square
Newcastle Helix
Newcastle upon Tyne, NE4 5TG
United Kingdom
Email: jdowland@redhat.com

Paul Watson
National Innovation Centre for Data
3 Science Square
Newcastle Helix
Newcastle upon Tyne, NE4 5TG
United Kingdom
Email: paul.watson@newcastle.ac.uk

Adam Cattermole
School of Computing, Newcastle University
Urban Sciences Building, 1 Science Square
Newcastle Helix
Newcastle upon Tyne, NE4 5TG
United Kingdom
Email: a.cattermole@newcastle.ac.uk

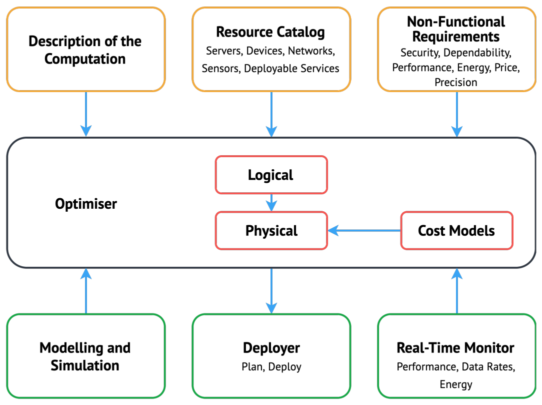
***Abstract*—We present a vision for the automatic optimisation of distributed stream processing programs. *StrIoT* — a distributed stream-processing framework built using purely-functional programming — enables a set of validated logical optimisation rules to generate a set of possible deployment plans. A cost model then filters and ranks the plans before the best is automatically deployed across the cloud and edge devices. We describe *StrIoT*'s functional operators for writing stream-processing programs; the design, implementation and performance of *StrIoT*'s logical optimiser; and the cost model, which filters and ranks re-written programs and deployment plans in terms of two non-functional requirements: bandwidth and cost. The *StrIoT* vision is being explored through an open-source proof-of-concept implementation. We present our initial results with a motivating example before outlining the success criteria for future work in this area.**

I. Introduction

Extracting value from streams of events generated by sensors and software is the key to success in many important problem domains, including wearable medical sensors, smart cities and the Industrial Internet. However, writing streaming data applications is not easy. Developers are confronted with major challenges, including processing events arriving at high rates, distributing processing over a set of heterogeneous platforms ranging from sensors to cloud servers, and meeting non-functional requirements such as energy, networking, security and performance. Current approaches leave almost all the responsibility for overcoming these challenges to application programmers.

We have been exploring an alternative approach to overcome these difficulties automatically. Our vision is of a system where developers write a stream-processing program in terms of a set of purely-functional operators. They focus on the functional requirements of their application, without conflating it with non-functional requirements such as deployment concerns. This is then combined with a description of the deployment environment and the non-functional requirements to which the application should be optimised (Figure 1). As the program is written in a purely functional framework, using a small set of operators that have clear, well-understood semantics, the application written by the programmer can be optimised to generate a set of possible deployment options. A cost model is then used to select the one that best meets the non-functional requirements. The set of optimisation rules we have developed have all been formally checked for correctness. This part of the system —- the focus of this paper — has been implemented and explored through *StrIoT* (*Str*eam processing for *IoT*)— an opensource proof-of-concept implementation, with the framework automatically deploying applications to the cloud and fog/edge devices.

II. *StrIoT*



Description of the Computation
Resource Catalog
Servers, Devices, Networks, Sensors, Deployable Services
Non-Functional Requirements
Security, Dependability, Performance, Energy, Price, Precision
Optimiser
Logical
Physical
Cost Models
Modelling and Simulation
Deployer
Plan, Deploy
Real-Time Monitor
Performance, Data Rates, Energy

Fig. 1. StrIoT system architecture

StrIoT is implemented in the purely-functional language *Haskell* [1] and available as open-source [2]. The source includes a range of example programs which can be built, optimised, deployed and executed. The run-time implementation makes use of Docker [3]-format containers to package the sub-programs for deployment.

The run-time and adaptivity aspects of the system have been described in [4]. This paper focusses on: the design of the operators provided for composing programs; the *Logical Optimiser* and the *Cost Models*.

We model a stream in *Haskell* as a (possibly infinite) list of events:

```
data Event a  = Event { time  :: Maybe Timestamp
                      , value :: Maybe a }

type Stream a = [Event a]
```

`Event` has been designed to be as general as possible: it can hold data of any type (e.g. integers, strings, tuples, lists, trees, graphs and even functions). An `Event` can also optionally hold a timestamp[1].

From the perspective of the operators described below, the `Stream` type is opaque: the user writes their program in terms of functions which operate on the payload data within a stream and are not directly burdened with manipulating the stream type itself.

A. Functional stream-processing operators

Based on an analysis of the literature on both stream processing and Complex Event Processing [5] and by experimenting with the implementation of a range of applications (available at [2]), we have provided application developers with the 8 pure stream-processing operators to compose their applications (described later in this section).

Our approach to designing the operators was to provide a balance between defining a small core set with the simple, clear semantics needed for analysis by our Logical Optimiser, described in Section III, while providing a sufficiently expressive programming environment for the application writer (by directly supporting the main operations found in stream processing applications).

We are confident that any stream-processing problem that can be described in a conventional, non-pure stream-processing system can also be implemented in *StrIoT*; A range of example programs is provided within the *StrIoT* source repository, including a solution [6] to the DEBS 2015 Grand Challenge [7].

We now give an overview of the stream-processing operators, and demonstrate their use in an example program which we build up, one operator at a time. This example is based on a real-world medical use-case from *PATH2iot* [8], a relational declarative stream-processing framework developed as part of a precursor project to *StrIoT*.

B. Filtering

Basic stateless filtering is achieved with `streamFilter` (there is a corresponding operator – `streamFilterAcc` – for stateful, history-based filtering). The user provides a predicate which operates on values from the input stream and returns a boolean to signal whether the event should be emitted on the output stream.

```
streamFilter :: (a -> Bool) -> Stream a -> Stream a
```

[1]`Maybe` is a Haskell datatype that can contain a value, or `Nothing`

For example, if a wearable device provided a set of sensor readings, we could filter the events to only those where a desired property held with a predicate function `vibrationModuleActive` provided by the user:

```
streamFilter vibrationModuleActive getWearableData
```

C. Mapping

The function `streamMap` is used to transform the values in a stream. The programmer supplies a transformation function which is applied to every event in the input stream.

```
streamMap :: (a -> b) -> Stream a -> Stream b
```

Continuing the previous example. Consider that the sensor readings include movement data from motion sensors and the user wishes to calculate the magnitude of the vector of movement by applying a Euclidean function: [2]

```
streamMap euclideanDistance
    $ streamFilter vibrationModuleActive
    $ getWearableData
```

`streamMap` is memoryless: the user-supplied function operates on a single event at a time and does not have access to the value of earlier events in the stream. Where it is necessary to take into account previous events, for example to build stateful aggregations, we provide `streamScan`:

```
streamScan :: (b -> a -> b) -> b -> Stream a
           -> Stream b
```

Here, the user-supplied function takes a second parameter, the *accumulator*. When the function is invoked, the value returned by the previous invocation of the `streamScan` is provided. In addition to the user-supplied function, the user also provides an initial value for the accumulator.

D. Filtering with memory

`streamFilter` is memoryless. For situations where knowledge of prior filtering decisions is required, we designed `streamFilterAcc`.

Much like `streamScan`, `streamFilterAcc` and the predicate function are extended to operate with an accumulator. Unlike `streamScan`, the accumulator value is not emitted on the output stream. The accumulator could, for example, be a list of previously seen values. In addition to the filter predicate, `streamFilterAcc` requires an accumulator update function to be supplied.

```
streamFilterAcc :: (b -> a -> b)
                -> b
                -> (a -> b -> Bool)
                -> Stream a
                -> Stream a
```

Continuing the running example. Suppose the user wishes to filter out events which describe a movement which is below a threshold relative to the previous event.

[2]The $ Haskell operator is used to sequence evaluation. When there is a series of functions separated by $, evaluation reads right-to-left.

```
streamFilterAcc (\_ new -> new) 0 thresholdCheck
    $ streamMap euclideanDistance
    $ streamFilter vibrationModuleActive
    $ getWearableData
```

In the above example, the accumulator update function simply returns the most recently seen value and the accumulator is initialised to `0`.

E. Windowing

`streamWindow` collects together incoming events and batches the data from them into lists to be emitted. `streamWindow` must be provided with a *window maker* function which implements the criteria for which events to group together.

The user can write their own window maker or use one of a set of common ones provided by *StrIoT*. These are: `sliding`, for overlapping windows of fixed length; `chop`, for fixed-length, non-overlapping windows; and two variants which operate on time intervals calculated from the timestamp field of incoming events (discarding any events without a timestamp): `slidingTime` and `chopTime`.

```
type WindowMaker a = Stream a -> [Stream a]
streamWindow :: WindowMaker a -> Stream a
             -> Stream [a]
```

In our running example, the user is now interested in collecting together batches of events that occur within a specified time interval. They can use the built-in window maker function `chopTime`:

```
streamWindow (chopTime 120)
  $ streamFilterAcc (\_ new -> new) 0 thresholdCheck
  $ streamMap euclideanDistance
  $ streamFilter vibrationModuleActive
  $ getWearableData
```

`streamExpand`, the dual of `streamWindow`, receives events which contain lists of some type and unpacks the list, emitting each individual item separately.

```
streamExpand :: Stream [a] -> Stream a
```

F. Combining Streams

Many stream-processing programs receive input from multiple sources and there is often a requirement to aggregate them together. `streamMerge` takes a list of stream inputs (of the same type), such as data from a series of IoT sensors, and interleaves their events into a single output stream.

```
streamMerge :: [Stream a] -> Stream a
```

The final operator, `streamJoin` (sometimes called `zip` in other systems) is used for combining exactly two inputs of different types. `streamJoin` pairs events from each input stream together and emits them as tuples.

```
streamJoin :: Stream a -> Stream b -> Stream (a,b)
```

III. Logical Optimiser

The *Logical Optimiser* is responsible for transforming the program supplied by the user in order to improve its performance with respect to the specified non-functional requirements, such as performing more quickly or at lower cost, whilst preserving its functional behaviour.

A. Rewrite rules

StrIoT is built in a purely-functional programming language, *Haskell*. Purely-functional expressions are *referentially transparent*, and can be substituted for any other expression which evaluates to the same value for the same inputs. This enables *equational reasoning*, a technique for transforming functions through a process of substitution by applying laws, or rules [9]. Rewrite rules have been successfully deployed as an optimisation tool within GHC [10], the principal *Haskell* compiler.

The combination of purely-functional semantics and the restricted set of operators with well-understood semantics enables the application of a rewrite system to logical optimisation.

1) Designing rewrite rules: In order to develop the rewrite system we systematically considered all 64 pairings of the 8 *StrIoT* operators. For each pair, we considered what transformation could be applied if that pairing of operators occurred in a stream-processing program: could we combine the operators together, or swap their order, or eliminate one or both?

```
streamFilter q . streamFilter p
 = streamFilter (\x -> p x && q x)

streamFilterAcc f a q . streamFilter p
 = streamFilterAcc (\a v-> if p v then f a v else a)
                   a (\v a -> p v && q v a)

streamFilter q . streamFilterAcc f a p
 = streamFilterAcc f a (\v a -> p v a && q v)

streamMap g . streamMap f = streamMap (g . f)

streamScan g a . streamMap f
 = streamScan (flip (g . flip f)) a
 where flip f a b = f b a
```

Fig. 2. Example of rewrite rules implementing Fusion

This process yielded 22 distinct, semantically-preserving rewrite rules for use within the rewrite system. A sample of rules is provided in Figure 2. During this process, we attempted to classify the rules according to established categories of stream-processing optimisations [11]. Of the ten categories, five are for logical optimisations, and we matched rules to three: *Operator re-ordering*, *Fusion* and *Operator separation*. We realised however that even if a given rule was not obviously advantageous on its own, by transforming the program it may open up a further rewrite opportunities from successive rule applications. We therefore kept these rules in our rule-set.

```
streamFilter f (streamMerge [s1, s2]) =
 streamMerge [streamFilter f s1, streamFilter f s2]
```

Fig. 3. The filter "hoisting" rule

In order to have assurance that the rules we derived were sound, we used the *QuickCheck* [12] tool to generate large volumes of test data to feed into operator pairings before and after applying each rule.

This approach helped us to discover that some rules we initially thought were sound actually altered the output stream.

For example, the rule in Figure 3 describes a case where a filter occurs downstream from multiple input streams merged together. The rule moves the filter upstream of the merge, duplicating it to each incoming stream. This could be beneficial by reducing the volume of events that need to be transmitted over an expensive network link.

By moving the filtering earlier in the stream, the order of events emitted by the merge operator differs: the sequence of events that arrive at the merge operator are different due to the filtering taking place. However, it is clear that minor changes in event ordering are unimportant for many practical stream processing systems. We designed seven such rules that could improve performance despite event re-ordering and we decided to make them optionally available to the application developer who wanted to take advantage of their performance-enhancing potential.

B. Rule Application

The Optimiser applies all matching rewrite rules to the supplied program, yielding a set of program variants. It then repeats this process for each variant. Since the rule-set doesn't guarantee convergence or termination, we limit the process to five successive applications. This was chosen by experimentation to provide a balance between deriving sufficient variants and the required processing time.

C. Partitioning

The types of non-functional requirement that we wished to support in *StrIoT* could depend upon properties of the distributed environment, such as the cost of the set of cloud nodes required to support the program or the bandwidth between edge and cloud devices. Our Cost Model (described in the next section) needs to consider not just the program but also the *deployment plan*: the mapping of operators from the program to the distributed nodes upon which it will be executed. We therefore needed to generate sets of deployment plans for each program variant, to be costed as a pair.

In our trial implementation we opted for a simple combinatorial approach (more scalable standard optimisation methods could be added later where this exhaustive approach is too expensive). The number of partitionings of a program consisting of n operators is therefore equal to the number of *compositions* of a positive integer: 2^{n-1}.

The Partitioner outputs pairings of program variants from the Logical Optimiser with each of the deployment plans as inputs for the Cost Model.

IV. Cost Models

The Cost Model is a critically important component of *StrIoT* as it is responsible for selecting the best pairing of program variant and deployment plan to satisfy the specified non-functional requirements.

The two non-functional requirements that we have initially focused on are deployment cost and bandwidth.

To achieve this we have applied queueing theory [13] [14] to derive a model from the input stream-processing program in order to predict properties of the program during execution.

To build this model we require information about properties of the program: For each source, a mean average event arrival rate (λ); For every operator, the mean average time taken to service each event (μ) and for each filter, the mean average selectivity (f).

In our current implementation we require the user to provide estimates for these properties. Future work will enable *StrIoT* to measure values for these properties at run-time and continually update them based on changing circumstances.

The vision for *StrIoT*'s architecture (Figure 1) includes a *Resource Catalog* describing properties of the deployment environment as inputs to the system. In our current implementation these properties are an *aggregate node utilisation limit* and an *inter-node bandwidth limit*.

A. Bandwidth

The event arrival rate for each operator is modelled by propagating the source arrival rate estimates and applying the selectivity estimates (f) for the program's filter operators.

The bandwidth between nodes in a deployment plan is calculated by combining this event arrival rate information with the size of individually serialised stream events for the data-types used at that junction in the program. This estimate is limited to fixed size data-types and cannot presently estimate the bandwidth of variable-length data such as lists.

We also apply a user-configurable weighting to represent the cost of starting up a network connection.

The cost model rejects any plans where this calculated inter-node bandwidth exceeds the user-specified limit.

B. Deployment cost

To calculate and minimise deployment cost, we first calculate the utilisation ρ of every operator in the program using the formula $\rho = \frac{\lambda}{\mu}$.

StrIoT uses this information to identify programs where any operator is determined to be over-utilised ($\rho > 1$). In this scenario, events are arriving at the operator at a faster rate than the operator can process them, and the queue of unprocessed data would grow indefinitely.

In addition to considering operators individually, *StrIoT* sums the operator utilisations for each node in a deployment plan. If the sum of utilisations for a node exceeds the user-specified threshold (e.g. 70% utilisation), the plan is not viable.

Any programs deemed not viable, including the original supplied by the user, are rejected. If no viable programs remain, the user is informed and must reconsider their approach, by either raising the user-specified thresholds or adjusting their program.

The remaining options are assigned a cost based on the number of nodes required for the deployment, with fewer nodes considered better, since this corresponds to commissioning, maintaining and paying for fewer cloud resources. The lowest cost option is chosen for deployment.

V. Illustrative Example

In this section we describe the performance of the Logical Optimiser and Cost Model applied to the running example program from Section II-A.

The complete example program is provided in Figure 4.

```
streamSink
  $ streamMap length
  $ streamWindow (chopTime 120)
  $ streamFilterAcc (\_ new -> new) 0 thresholdCheck
  $ streamMap intSqrt
  $ streamMap euclideanDistance
  $ streamFilter vibrationModuleActive
  $ getWearableData
```

Fig. 4. Example *PATH2iot* program

The developer supplied the additional information outlined in Section IV: an inter-node bandwidth limit of 30 kB/s and an aggregate node utilisation limit of 90%.

A. *PATH2iot program*

Let us first explore the performance of the unmodified input program. The Cost Model is applied to each pairing of the program with one of 127 potential deployment plans generated by the Partitioner.

The Cost Model calculated the estimated bandwidth required at the egress of every operator in the program. This was determined to exceed the user specified threshold of 30kB/s until the `streamWindow` operator, which occurs relatively late in processing. All plans which placed a node boundary prior to the window operator were therefore ruled out on the basis of the bandwidth requirement.

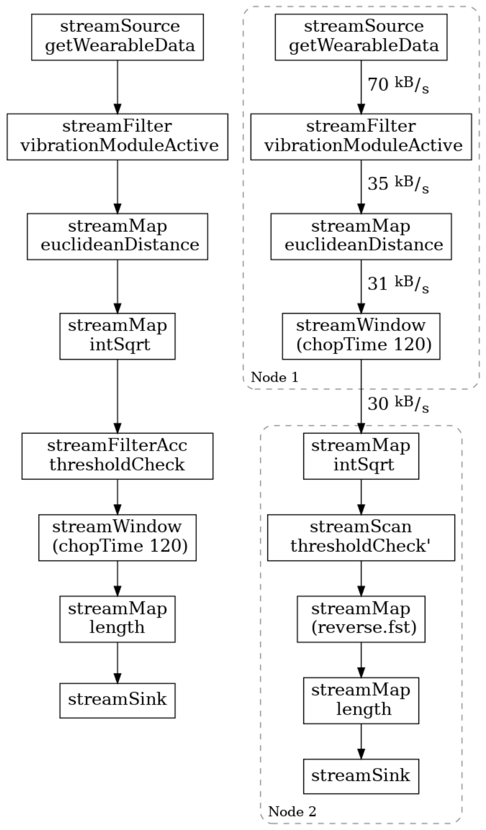
The remaining plans necessarily place all the operators prior to the window onto a single node. The total utilisation of all the operators placed on that node exceeds the user-specified aggregate node utilisation limit of 90%, and are also rejected.

StrIoT has thus determined that there are no valid deployments for the original program under the constraints supplied by the user.

B. *Logical Optimiser*

Now we analyse the behaviour of the Logical Optimiser applied to the program. 57 program variants are derived. The Partitioner then generates potential deployment plans for these variants, emitting 5,718 deployment options to be filtered and ranked by the Cost Model.

From these deployment options, all but 120 are determined unviable and rejected by the Cost Model. A final 8 are calculated to have the lowest deployment cost. One of these plans is depicted on the right-hand side of Figure 5. For comparison, the original program is depicted on the left-hand side.



streamSource getWearableData
streamFilter vibrationModuleActive
streamMap euclideanDistance
streamMap intSqrt
streamFilterAcc thresholdCheck
streamWindow (chopTime 120)
streamMap length
streamSink
streamSource getWearableData
70 kB/s
streamFilter vibrationModuleActive
35 kB/s
streamMap euclideanDistance
31 kB/s
streamWindow (chopTime 120)
Node 1
30 kB/s
streamMap intSqrt
streamScan thresholdCheck'
streamMap (reverse.fst)
streamMap length
streamSink
Node 2

Fig. 5. *PATH2iot* program, before (left) and after rewriting and partitioning (right). Edge labels indicate the estimated bandwidth between operators.

Each of the best-scoring plans have the same fundamental transformation: the `streamWindow` has been moved earlier in processing by two specific rewrite rules, illustrated in Figure 6.

```
streamWindow w . streamFilterAcc f a p
 = streamMap (reverse.fst)
 . streamScan (\ (_,acc) a -> filterAcc f p acc a)
    ([],a)
 . streamWindow w

streamWindow w . streamMap f
 = streamMap (map f) . streamWindow w
```

Fig. 6. Rewrite rules which move `streamWindow`

The bandwidth required after the newly-positioned window is calculated to be 30 kB/s, meeting but not exceeding the user-specified limit. The chosen deployment plan requires two nodes and places the node boundary at this point.

The Partitioner produced some alternative two-node deployment plans which placed the node boundary at an earlier point. These plans were rejected by the Cost Model as the bandwidth at those points exceeds the user-specified limit.

Plans which place the boundary later in processing, where the bandwidth is determined to be even lower, result in the first node (corresponding to the processing deployed on the sensor) doing too much work: the sum of the utilisations of the operators assigned to that node exceed the user-supplied maximum node utilisation, and are likewise rejected.

VI. Conclusion

Our main achievement to date has been to demonstrate, via an end-to-end proof of concept implementation, that the architectural vision for a purely-functional stream-processing system (depicted in Figure 1) is viable. We have made our implementation available as open-source software [2] and included several examples of stream-processing programs that can be built, optimised and deployed.

The set of stream-processing operators have been carefully designed to strike a balance between providing an expressive environment for the developer and having clear and well-understood semantics for formal analysis.

The focus of this paper has been on the Logical Optimiser. Our chosen design takes advantage of referential transparency provided by the purely-functional semantics of *Haskell*, combined with restricting users to the specific set of operators with well-understood semantics, to build a rule-based rewriting system. We have demonstrated that this is a valid approach for an optimiser and there is merit in continuing to fully exploring its potential.

Although not the focus of this paper, *StrIoT* can also exploit the clear semantics to enable run-time adaptivity, including moving operators between the edge and the cloud. This is described in [4].

One of the main outcomes from the work on designing rewrite rules for the logical optimiser has been discovering that rather than strictly preserving all of the semantics of the stream-processing program, it is sometimes advantageous to alter them in well-defined ways, such as by permitting some re-ordering of stream events. We believe it is therefore a good idea for systems to permit users to clearly indicate which aspects of the semantics of their program are important, and which can be altered and to what extent.

VII. Success criteria and further work

Our next step is to refine our proof-of-concept, completing the implementation of the architecture depicted in Figure 1 and conduct further experimental evaluations on a range of applications to demonstrate the efficacy of our approach.

Our proof-of-concept contains a run-time component (described in more detail in [4]) but there is limited integration between the Logical Optimiser and run-time components. Run-time measurements could be used to provide values for the properties of the cost-model as described in Section IV, freeing the user from the burden of supplying estimated values themselves. The existing support for run-time adaptivity can be extended to include re-running the logical optimiser and deploying an alternative deployment which better meets the changed circumstances. Future work will also explore extending the run-time for flow control and fault tolerance.

The current modelling of the deployment environment assumes homogeneous nodes. Extending the Resource Catalog to describe a range of different types of nodes for deployment will allow the Cost Model to be extended to better support optimising for more situations, for example edge nodes being unable to perform work that requires hardware such as an FPU or GPU, or describing a wider range of the possible cloud node deployments and their cost.

Acknowledgments

Jonathan Dowland acknowledges Red Hat, Inc for supporting his contributions to this work. Adam Cattermole was supported by the Engineering and Physical Sciences Research Council, Centre for Doctoral Training in Cloud Computing for Big Data [grant number EP/L015358/1].

References

[1] S. Marlow, "Haskell 2010 language report," The Haskell community, Tech. Rep., 2010. [Online]. Available: http://www.haskell.org/onlinereport/haskell2010/

[2] *StrIoT* authors, "*StrIoT*," 2022. [Online]. Available: https://github.com/striot/striot

[3] D. Inc., "Docker," 2013. [Online]. Available: https://www.docker.com/

[4] A. Cattermole, J. Dowland, and P. Watson, "Run-time adaptation of stream processing spanning the cloud and the edge," in *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion*, ser. UCC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3492323.3495627

[5] M. Dayarathna and S. Perera, "Recent advancements in event processing," *ACM Comput. Surv.*, vol. 51, no. 2, pp. 33:1–33:36, Feb. 2018. [Online]. Available: http://doi.acm.org/10.1145/3170432

[6] *StrIoT* authors, "*StrIoT* solution to the debs '15 grand challenge [7]," 2022. [Online]. Available: https://github.com/striot/striot/blob/2022-08-24/examples/taxi/Taxi.hs

[7] Z. Jerzak and H. Ziekow, "The debs 2015 grand challenge," in *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '15. New York, NY, USA: ACM, 2015, pp. 266–268. [Online]. Available: http://doi.acm.org/10.1145/2675743.2772598

[8] P. Michalák and P. Watson, "Path2iot: A holistic, distributed stream processing system," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, IEEE. New Jersey, NJ, USA: IEEE, 2017, pp. 25–32.

[9] R. Bird, *Thinking Functionally with Haskell*. Cambridge University Press, 2014.

[10] S. Peyton Jones, A. Tolmach, and T. Hoare, "Playing by the rules: rewriting as a practical optimisation technique in ghc," in *2001 Haskell Workshop*, ACM SIGPLAN. New York, NY, USA: ACM, September 2001, pp. 203–233.

[11] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Computing Surveys (CSUR)*, vol. 46, 03 2014.

[12] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: Association for Computing Machinery, 2000, p. 268–279. [Online]. Available: https://doi.org/10.1145/351240.351266

[13] I. Mitrani, *Probabilistic Modelling*. Cambridge University Press, 1997.

[14] T. Z. J. Fu, J. Ding, R. T. B. Ma, M. Winslett, Y. Yang, and Z. Zhang, "Drs: Dynamic resource scheduling for real-time analytics over fast streams," 2015.