Declarative Distributed Stream Processing

PhD Stage 2 Report

Jonathan Dowland <jon.dowland@ncl.ac.uk>

June 2021

I have been studying part-time towards a PhD since 2017/18. This report marks the end of Stage 2 (my 4th year of study). In this report I outline the work completed during this stage, the work remaining and the plan for continuing to completion.

1 Background (2 pages)

1.1 Distributed Stream-Processing

Many modern applications, in domains ranging from smart cities to healthcare, have a requirement for the timely processing of data arriving at high speed, for example that generated by sensors in the Internet of Things (IoT). Such systems may need to meet a range of other requirements, including: reliability; security; energy efficiency, for example to prolong the battery life of sensors in the field; or privacy, to remove or de-personalise data prior to transmitting it over open networks.

The combination of requirements, very high data arrival rates, and the desire for timely processing, makes the design and management of the supporting infrastructure very challenging. The current generation of IoT tools adopt the principles of stream processing and are designed around a three-tier architecture: sensors generate data, which is sent on to a local gateway (e.g. a smartphone or embedded device) to be collated before being passed on to the Cloud for processing.

However, in some domains it can be beneficial to perform some processing on the gateway or on the sensors themselves[MW17], to reduce the volume of data sent onwards to the cloud; or to reduce the frequency with which sensors must invoke their networking hardware, thus reducing energy expenditure; or to avoid transmitting sensitive data across public networks, by filtering or anonymising data sets at the point of collection.

1.2 Purely-Functional Programming

Functional Programming (FP) is a software development paradigm where the principle building blocks of programs are functions[Bir14] (as opposed to e.g. abstract objects in Object-Oriented Programming) and programs are composed declaratively using expressions, rather than imperatively with sequences of statements.

Advocates of FP believe that many of its properties have advantages for the design and implementation of large and complex software systems[Cup89]. Constructing systems declaratively results in the programmer describing *what* a system should do, rather than the minutiae of *how* the work should be performed.

Purely-functional programming is a variant of FP where the behaviour of functions is entirely defined in terms of their input arguments and output value and can perform no

other actions (referred to as *side-effects*). Consequently, purely-functional expressions are *referentially transparent*, and can be substituted for any other expression which evaluates to the same value for the same inputs.

Referential transparency enables *equational reasoning*, a technique for transforming functions through a process of substitution by applying laws or rules[Bir14].

1.3 Purely-Functional Stream Processing

My research aim is to establish to what extent the advantages of purely-functional programming can be applied to the design and operation of a distributed stream-processing system.

Modern distributed stream-processing systems attempt to separate the functional definition (typically specified as a software program) and the non-functional requirements (the deployment environment and constraints such as power requirements, network utilisation limits, etc.). The declarative nature of purely-functional programming allows for a high degree of abstraction. I will investigate whether this enables the construction of a system where the user can specify the functional behaviour of the program declaratively and independent of the deployment and non-functional requirements.

In order to meet non-functional requirements, it may be necessary to adjust the stream-processing specification provided by the user whilst preserving the functional behaviour. Equational reasoning is a powerful tool for encoding program transformations and can be used to build rewriting systems[PTH01]. I will investigate whether rewrite rules are expressive enough to encode and apply useful transformations for stream-processing optimisation.

A distributed stream-processing system needs to partition and distribute a stream-processing definition onto individual nodes. I will investigate whether any facets of purely-functional programming are particularly beneficial for the design and implementation of the Partitioner, and how automated partitioning should interact with the Optimiser.

1.4 Foundations

We are exploring an alternative approach: a system whereby the stream processing operations and the non-functional requirements are described declaratively as inputs to an Optimiser, which automatically determines the most appropriate deployment onto the available resources, which may include sensors and gateways. Monitoring of the deployment is used to evaluate the performance of the Optimiser and could also be used for run-time adaptation. The initial approach (by PhD student Peter Michalák[MW17]) used an extended version of SQL as the method of describing the computation.

In contrast, in our project we are exploring using functional programming to describe the computation. Using the pure functional language Haskell, a prototype has been developed named "StrIoT"[aut20] where the stream processing is defined in terms of a restricted set of functional operators with well-understood semantics. This prototype has two distinct components: Library code to support stream processing in which segments of the stream are spread across multiple compute nodes; and the Optimiser.

My research will be focussed on the optimiser and deployer (another PhD student — Adam Cattermole — is working on the stream processing library).

A high-level overview of the StrIoT architecture is provided in Figure 1.

2 Work completed (1 page)

2.1 2019-20

The work in this section was completed in academic year 2019-20 and fully described in my Stage 2/Year 3 interim report[Dow20]. What follows here is a brief summary.

I further considered the rewrite rules described in [Dow19a] and matched some of them to known categories of stream-processing optimisation[Hir+14]. I recognised that some rules caused re-ordering of the stream of events and so were not functionally equivalent, although they could be useful in situations where re-ordering was not important.

I devised a scheme for encoding program rewriting rules as regular Haskell functions and implemented them in *StrIoT*. I designed and implemented an algorithm to apply rules to a stream-processing program. Since the rules do not form a terminating system, the algorithm applies rules successively up to a user-supplied threshold and then stops.

I spent some time exploring the Template Haskell[SP02] meta- programming system in order to reason about the parameters supplied to *StrIoT* operators by the user (e.g. filter predicates). I determined that this was impractical, although I did manage to apply Template Haskell to improve *StrIoT*ś ease of use.

Automatic partitioning of the stream-processing program. I designed and implemented an algorithm to generate a list of all possible partitionings of a stream-processing program. The algorithm encodes a number of limitations on the way in which operators could be distributed to physical nodes: for example, a "merge" operator must be placed as the first operator in a Node; A source operator and a sink operator cannot co-exist on the same Node.

DEBS 2020 paper. Paul Watson, Adam Cattermole and I wrote a paper describing the work on *StrIoT* to-date and submitted it to the 14th ACM International Conference on Distributed and Event-based Systems [Var20]. The paper was rejected but we received very constructive feedback from the reviewers which has informed my plans for the work remaining.

2.2 Engineering

Much of the last calendar year was spent on further *StrIoT* engineering. The existing user interface had a number of problems that needed addressing: the user was required to supply a partition-map alongside the stream-processing program; this partition map necessarily referenced the operators in the program prior to any optimisation. Optimisation could add or remove operators, invalidating the partition map. To avoid this problem *StrIoT* applied optimisation to the sub-programs after performing partitioning. This limited the extent to which the optimisation process could rewrite the program and consequently the impact rewriting could make on its performance. As noted in the review feedback from our DEBS '20 paper, *StrIoT* lacked a cost model and so could not evaluate the performance of optimisation beyond a very simplistic heuristic such as minimising the number of operators.

I developed a new user-interface for *StrIoT* to which a use need only supply the stream-processing program and not a hand-written partitioning scheme. Instead, we apply the automatic partitioning to every rewritten program obtained by applying the rewrite rules, to obtain a list of program/partition-map pairings. I then developed the framework for a cost model function which operates on such pairings.

2.3 Cost Model

I spent considerable time studying queuing systems and networks in order to apply them in developing a cost model for evaluating the performance of stream-processing programs. I am exploring whether a stream-processing program can be modelled as a queuing network; properties of the program (and its constituent operators) calculated and those properties used as part of the costing.

I extended the data types within *StrIoT* to include parameters used in the queuing theory calculations: arrival rates for sources, selectivities for filters and average service times for all operators. The logical optimiser in particular required significant re-engineering to account for queuing theory parameters in rewrite rules. Careful decision-making was needed to answer questions such as: what should the selectivity be of a fused filter operator?

I extended proof-of-concept code implementing queuing theory calculations to operate on the *StrIoT* data-types. Finally I wrote an initial cost model function which applied queuing theory calculations to determine the utilisation of all operators within a stream-processing program and attempted to minimise the overall utilisation.

3 Work remaining (2 pages)

3.1 Revised DEBS paper

The paper we submitted to DEBS 2020 was rejected but we received some very helpful feedback from the reviewers. I will re-work the paper, address the concerns raised by the reviewers and submit to a future conference.

3.1.1 Improved cost model

The most significant problem is the lack of a cost model to demonstrate the effectiveness of our approach to stream-processing.

I am designing and implementing a cost model based on queuing theory that, when paired with an appropriate stream-processing program, should yield an interesting result that demonstrates the value of our approach.

The initial improved cost model will attempt to apply Bin Packing to output a deployment plan that maps logical operators to physical nodes in an attempt to minimize the number of nodes required.

We derive a queuing theory model from the stream-processing program where each operator is represented as a Server. From that model we calculate their individual utilisations.

The plan maps logical operators to physical nodes such that the maximum number of operators are assigned to each node without breaching a maximum aggregate utility threshold which we will choose from experimentation.

3.1.2 Consistent example problem

We did not use a single example problem throughout the paper. We opened with simple examples in the IoT domain, but switched to alternative problems for later sections. This was identified as a source of confusion by reviewers.

Our *StrIoT* implementation also lacks an encoding of a stream-processing program for a real-world problem which is demonstrably improved by the rewriting or partitioning processes.

I will devise an example stream-processing program within a real-world problem domain such that processing it with *StrIoT* results in a clearly improved program (in terms of a non-functional requirement of relevance to the domain). I will then use this example program consistently throughout the revised paper.

3.1.3 Design justification

We did not sufficiently explain the rationale behind some of our design choices, such as the use of purely-functional programming, Haskell as the implementation language, or the choice of the restricted operators from which users can compose their programs. Some reviewers were unsure what was novel about our approach, pointing out that the choice of operators are common to many existing systems.

Some of these issues can be addressed by revising the existing material to make justifications more clear. For example, our Logical Optimiser is only possible due to the choice of a purely-functional language.

I will expand the comparison to existing stream-processing systems in use in the field. To support this I will revisit surveying all such systems and their relevant properties.

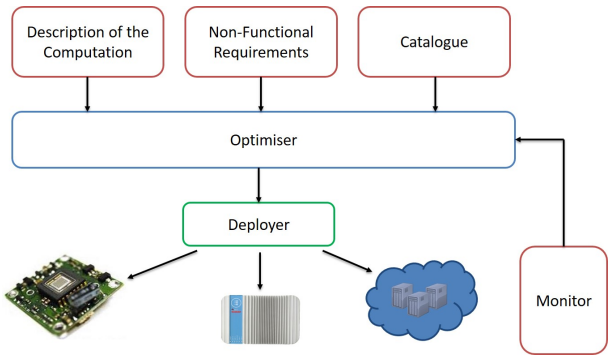

Figure 1: StrIoT architecture diagram

3.2 Further cost models

The immediate plans for a Cost Model are relatively modest and intended to quickly demonstrate the efficacy of *StrIoT* for the revised DEBS paper. Further refined models could be developed to reason about other non-functional requirements or to address whether the assumptions of the queuing theory calculations hold.

Other non-functional requirements include constraints on bandwidth between physical nodes by considering the size of event data in conjunction with the rate of events being transferred.

The plan relies upon a queuing theory model which has a set of assumptions that must hold for the calculations to be valid. These include constraints on the distribution of arrival rates of events throughout the network. Some of the stream-processing operators, such as filters and windowing, can break these assumptions. Further work may include exploring chains of queuing networks, or Kingsman correction,

3.3 Paper on derivation of rewrite rules

I plan to write a separate paper based on the work described in [Dow19a], Section "Derivation of stream rewriting rules" and encoding and categorisation described in [Dow20], Section 2.1.

Throughout the last Stage I prioritised preparing the DEBS paper and the elements of my work that were required to support it. Work on a rewrites paper was on hiatus. I did return to my notes on rewrite rules from time to time as and when my main worked touched on that area or when I had a realisation about a rewrite rule or their properties.

The most significant new work on rewrite rules was beginning to map each rule to a category of known stream-processing optimisations[Hir+14]; describe where some rewrite rules cause re-ordering of the input events and recognising that the flexibility afforded to user-supplied windowing functions made analysis of windowing difficult.

One area to expand on is to begin analysis of specific window-maker functions supplied with *StrIoT* (sliding or non-overlapping windows based on event times or number of events) as we can reason further about their behaviour than other unknown, user-supplied functions.

3.3.1 QuickSpec

QuickSpec[Sma+17] is a system for discovering rules and laws from a set of pure functions. It would be interesting to see whether QuickSpec or a similar tool could be used to

derive further logical rewrite rules that I did not discover in my systematic operator comparison[Dow19a].

4 Plan (2 pages)

My main focus in the last Stage was the preparation of our Paper for DEBS 2020 ("Paper #1" in my original plan[Dow19a]). I found it especially effective (and motivating) to organise my work towards that milestone. I'm therefore structuring my remaining work around similar milestones which are described below.

The broad outline of my plan is summarized in the GANTT chart in Figure 2.

4.1 Plan review

When I wrote my Plan in June 2019 I tried to consider all possible risks to the project, but I failed to consider an international Pandemic, which has had a significant impact on my work. For health reasons, following Government advice, I spent most of 2020 isolating together with my immediate family. I am very fortunate that I was able to continue PhD work during this time, although it was (and remains) very difficult to predict how much time I will be able to assign to PhD studies on a week-by-week basis.

By this point in time, my original plan forecast that Implementation and Testing work would be complete, and two papers written.

Implementation is largely complete, although I expect a continued stream of low-level maintenance work and bug fixing to continue throughout the remaining time. An initial Cost Model is in place. The bulk of any further implementation work will be on improved cost models.

I wrote and submitted one of the two forecast papers, and opted to focus on that work and pause work on the second paper.

4.2 Milestones

4.2.1 Revised DEBS paper

This milestone collects the majority of the work required for the core of my PhD. An important initial task is to select the conferences or journals to target and determine the relevant deadlines. Assuming we target DEBS 2022: Submission deadline is usually February/March for the conference taking place the following June/July. Working on this basis I have up to approximately 6 months (Jul 2021 to Feb 2022) to complete the supporting work and write the paper.

4.2.2 Rewrite rules paper

As discussed in Section 3.3, I plan to write up my work on program rewriting rules. This is a lower priority than the revised DEBS paper as the relevant work that contributes to the core of my PhD is already complete. I have enough material to form the basis of this paper, but depending on the rate of progress of my core objectives, and whether I choose to pursue an extension activity in the area of rewrite rules, the scope of this paper could be expanded.

In common with the revised DEBS paper, I shall first identify the conferences or journals I plan to submit to, in order to establish a submission deadline to work to.

4.2.3 Further cost models

As described in Section 3.2, there are a number of avenues for improving the cost model. Which route to take depends upon how well work goes on the initial cost model and

queuing theory and whether our simple cost model to be described in the revised DEBS paper is sufficient to publish some results.

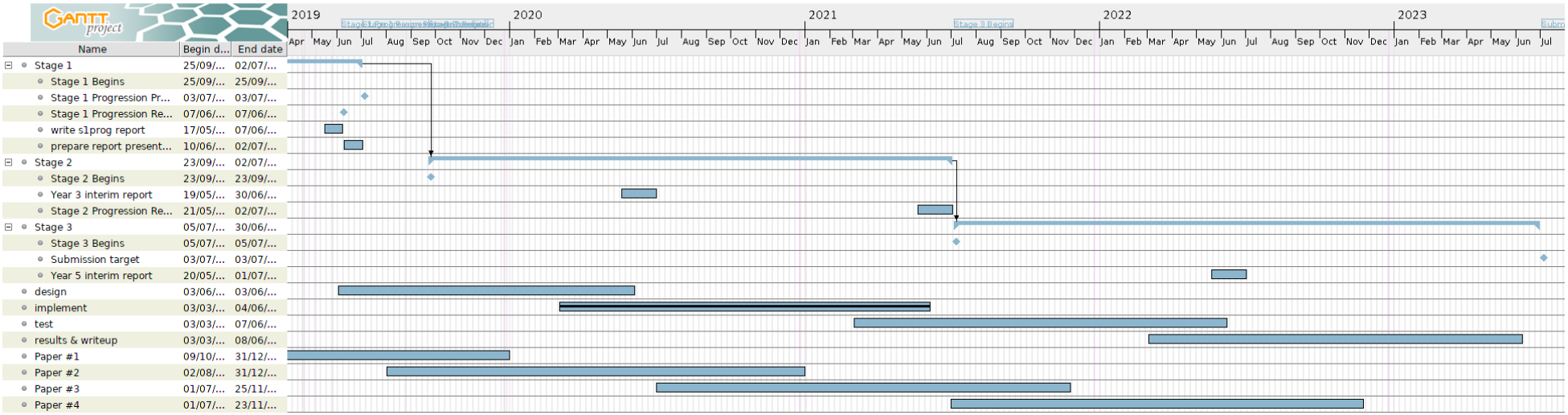

Figure 2: Project GANTT Chart

5 Thesis Outline (1 page)

Early on in my research I focussed on the logical optimiser aspect of the architecture and attempting to prove its value has become the focus of my PhD.

The rewrite rules, their derivation, analysis and applying them to stream-processing programs is one main piece of work that is complemented by demonstrating their effectiveness with a robust cost model grounded in a sound theory.

My draft thesis outline below reflects the importance of those two main aspects of my work.

5.1 Draft outline

1. Abstract
2. List of Research Outputs
3. Background
   - 3.1. Stream-processing systems
   - 3.2. Purely-Functional programming
   - 3.3. StrIoT architecture overview
   - 3.4. Queuing theory
4. Program rewriting
5. Cost Model
6. Results
7. Conclusion
   - 7.1. Thesis summary
   - 7.2. Contributions
   - 7.3. Future research directions
8. Appendices

6 Research Outputs (1 page)

6.1 DEBS 2020 paper submission

Paul Watson (my supervisor), Adam Cattermole and I wrote and submitted a paper to the 14th ACM International Conference on Distributed and Event-based Systems [Var20].

The paper is an overview of the *StrIoT* system and work completed to-date. My main contribution is the section on logical optimisation and deployment.

During the process of writing the paper we decided that we would be unable to complete the work needed to implement a Cost Model into our proof-of-concept system in the time available to us before the submission deadline. For this reason we instead focussed on ensuring the other components were completed and functioning to a high standard.

6.2 *StrIoT* open source software

The primary output from my work so far is the research stream-processing platform *StrIoT*, which is open source software. I have made many contributions to *StrIoT* within this stage, which have been organised into 36 separate GitHub "Pull Requests"[Dow21b].

6.3 Personal blog posts

In order to practice technical writing, explore how to summarize and present the area of focus at a given time and to get early feedback from relevant online communities, I have been blogging about my PhD work. In this Stage I have written ten blog posts[Dow21a]. These have attracted some useful comments and suggestions in particular from the Haskell functional-programming community. On three occasions my blog posts were picked up and re-broadcast by "Haskell Weekly News", a popular community newsletter.

6.4 PhD Poster Session

I prepared and submitted a poster to the School of Computing PhD candidates poster session in 2019[Dow19b]. I adopted the "Better Poster" format[Mor19], substituting the "main finding" as the aspect given the highest priority for my research question, which is more fitting for the stage of my research.

6.5 Short presentation for Dr. Paul Ezhilchelvan

I began working with Dr. Paul Ezhilchelvan as I started exploring Queuing Theory as the theoretical underpinnings for a cost model. I prepared a short, specific presentation for Dr. Ezhilchelvan which aimed to provide a brief introduction to the design of *StrIoT* and how I was attempting to its concepts into queuing theory.

References

[aut20] *StrIoT* authors. *StrIoT*. 2020. URL: https://github.com/striot/striot.

[Bir14] R. Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2014. ISBN: 9781107087200. URL: https://books.google.co.uk/books?id=B4RxBAAAQBAJ.

[Cup89] John R. G. Cupitt. "The Design and Implementation of an Operating System in a Functional Language (Miranda)". AAIDX92465. PhD thesis. Canterbury, UK: University of Kent at Canterbury, 1989.

[Dow19a] Jonathan Dowland. *Declarative Distributed Stream Processing. PhD Stage 1 Progression Report*. 2019. URL: https://jmtd.net/log/phd/dowland_phd_stage1_progression_report.pdf.

[Dow19b] Jonathan Dowland. *PhD Poster*. 2019. URL: https://jmtd.net/phd/poster/.

[Dow20] Jonathan Dowland. *Declarative Distributed Stream Processing. PhD Stage 2 Year 1 Report*. 2020. URL: https://jmtd.net/log/phd_year_3_progression/.

[Dow21a] Jonathan Dowland. *PhD blog posts in Stage 2*. 2021. URL: https://jmtd.net/phd/stage2/.

[Dow21b] Jonathan Dowland. *StrIoT Pull Requests by Jonathan Dowland between 2019-07-01 and 2021-07-01*. 2021. URL: https://github.com/striot/striot/pulls?q=is%3Apr+is%3Amerged+created%3A2019-07-01..2021-07-01+author%3Ajmtd.

[Hir+14] Martin Hirzel et al. "A Catalog of Stream Processing Optimizations". In: *ACM Computing Surveys (CSUR)* 46 (Mar. 2014). DOI: 10.1145/2528412.

[Mor19] M. A. Morrison. *#betterposter*. 2019. URL: https://osf.io/ef53g.

[MW17] Peter Michalák and Paul Watson. "PATH2iot: A Holistic, Distributed Stream Processing System". In: *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2017, pp. 25–32.

[PTH01] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. "Playing by the rules: rewriting as a practical optimisation technique in GHC". In: ACM SIGPLAN. Sept. 2001. URL: https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/.

[Sma+17] Nicholas Smallbone et al. "Quick specifications for the busy programmer". In: *Journal of Functional Programming* 27 (2017), e18. DOI: 10.1017/S0956796817000090.

[SP02] Tim Sheard and Simon Peyton Jones. "Template meta-programming for Haskell". In: Oct. 2002, pp. 1–16. URL: https://www.microsoft.com/en-us/research/publication/template-meta-programming-for-haskell/.

[Var20] Various. *14th ACM International Conference on Distributed and Event-based Systems*. 2020. URL: https://2020.debs.org/.