Functional HDLs Meet Stream-based Monitoring: Hardware Monitors from RTLola Using Clash

Saarland University

Department of Computer Science

MASTER'S THESIS

submitted by

Bipin Oli

Saarbrücken, July, 2025

Supervisor:	Prof. Bernd Finkbeiner, Ph.D.
Advisor:	Jan Eric Baumeister, M.Sc.
Reviewers:	Prof. Bernd Finkbeiner, Ph.D. Felix Klein, Ph.D.
Submission :	31.07.2025

Abstract

Many critical systems require real-time monitoring to ensure correct operation, making the correctness of runtime monitors essential. Monitoring languages like RTLola provide formal semantics over input and output streams to support this. These monitors must operate with minimal resource overhead and speed, often necessitating hardware-based implementations. While RTLola has an existing hardware architecture and VHDL-based compiler, VHDL lacks the expressiveness and type safety of functional HDLs like Clash. Clash, based on Haskell, enables more maintainable and semantically rich monitor specifications, including direct enforcement of RTLola semantics such as pacing. Moreover, it supports multiple HDL backends, offering flexibility beyond VHDL. The current RTLola hardware architecture also lacks pipelined stream evaluation, limiting performance. In this work, we extended the RTLola hardware architecture to support pipelined evaluation and developed a Clash-based compiler backend that incorporates key RTLola semantics directly into the generated HDL, enhancing performance, safety, and portability.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Jan Baumeister, for his valuable guidance, and mentorship throughout this research journey. I am deeply grateful to my supervisor, Prof. Finkbeiner, for introducing me to this fascinating topic and for reviewing this thesis. My appreciation also extends to Felix Klein for his review. I wish to also thank the Clash community for their helpful responses to my many questions on the community Slack and Discourse. And finally, I am thankful to Lisa for her care and the many bowls of yoghurt with strawberries, and to my parents for their constant support.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

Declaration of Consent I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, 31 July, 2025

Erklärung

Ich erkläre hiermit, dass die vorliegende Arbeit mit der elektronischen Version übereinstimmt.

Statement

I hereby confirm the congruence of the contents of the printed data and the electronic version of the thesis.

Saarbrücken, 31 July, 2025

Contents

Chapter 1

Introduction

Imagine a swarm of tiny medical robots navigating through a patient's blood vessels, filtering out toxins, clearing fatty clogs, and operating on inner wounds. These robots must autonomously adapt to their surroundings and respond to sudden environmental changes, such as fluctuations in blood pressure caused by the heartbeat while precisely operating on the target. Imagine a high-frequency trading system that monitors fluctuations in the market to identify marginal upsides — and reacts as quickly as possible before the window of opportunity disappears. Or consider a missile defense system that tracks the position of incoming threats and launches counter-missiles with the right angle and velocity, all while accounting for dynamic factors like wind currents and moving targets. All such critical systems have one thing in common: the need to continuously monitor themselves in real time as they operate, detect deviations as they arise, and perform corrective actions in time — before it is too late.

Depending on the acceptable level of latency, such monitors can be implemented in various ways. On a digital computing platform, one might run the monitoring logic on a real-time operating system. If that proves insufficient, a bare-metal implementation on the CPU may be necessary. However, this approach is still constrained by the inherently sequential nature of the CPU's instruction fetch-decode-execute cycle.

Importantly, speed is not the only concern. Many systems are tightly constrained in terms of other resources such as energy consumption, physical area, and weight. Consider, for example, a monitoring component embedded in those tiny medical robots. In such a scenario, it is critical that the monitor be both fast and extremely resource-efficient.

This is where hardware monitors become appealing: by designing custom chips tailored to the specific monitoring task, one can achieve significantly higher parallelism and performance, often with substantially lower resource footprints.

Expressing the monitoring requirements, however, presents a different challenge. General-purpose programming languages, while powerful and expressive, are often ill-suited for specifying monitoring logic. Their versatility comes at the cost of formal semantics, making it difficult to reason about correctness or to statically analyze monitoring behavior. After all, we would not want to end up needing a runtime monitor to verify our monitor. To ensure reliability and analyzability, it is important to specify monitoring requirements within a well-defined formal semantics.

To address this, a number of specification languages for runtime monitoring have been developed. Early efforts focused on logic-based formalisms. Later, Lola [1], [2] pioneered a stream-based approach, allowing users to express monitoring constraints declaratively in a dataflow-like manner. However, these early languages typically assumed a synchronous input model, which limited their applicability. After all, many real-world monitoring system must observe events that can occur independently and asynchronously. To overcome this limitation, asynchronous stream-based languages such as RTLola [3], TeSSLa [4], and Striver [5] were introduced, extending the expressiveness and flexibility of runtime monitoring in real-world systems.

From a hardware design perspective, designing complex hardware directly as circuit schematics can be extremely challenging. To address this, hardware description languages (HDLs) like VHDL, Verilog, and SystemVerilog are widely used in the industry. These languages allow designers to describe hardware at a higher abstraction level using code. The HDL code is then processed by synthesis toolchains, which perform optimization and various transformation steps to generate the final circuit schematic or configuration. This output can either be used for fabrication or programmed directly into programmable hardwares such as FPGAs.

While VHDL, Verilog, and SystemVerilog remain the industry standards with the most widely available toolchains and support, the HDL landscape extends beyond these traditional languages. For example, Clash is a functional hardware description language based on Haskell that enables more expressive hardware design. Clash can generate standard HDL outputs like Verilog, VHDL, or SystemVerilog, allowing seamless integration with existing synthesis toolchains.

By leveraging Haskell's strong static type system and expressive power, Clash brings enhanced type safety and abstraction to hardware design without sacrificing synthesizability. Despite its high-level nature, Clash still provides low-level control when needed; its templating system enables precise mapping to backend HDLs such as VHDL. As a functional HDL, Clash also aligns closely with declarative specification languages like RTLola.

When building a hardware-based monitor, there are two main implementation approaches: one is to design a general-purpose monitoring chip that can be programmed with specific specifications, and the other is to create specialized hardware tailored to a single, fixed monitoring specification.

From a certification point, the general monitor might be more advantageous. While it may be possible to formally certify the compilation process that generates a specific monitor, it is often more desirable to certify a general-purpose chip that can be programmed with a wide range of specifications. This approach simplifies certification by covering many use cases with a single hardware device. Additionally, when targeting ASICs, the lengthy and expensive fabrication process makes rapid prototyping of spec-specific hardware impractical. In such cases, FPGAs offer a practical alternative, enabling fast prototyping and iteration before committing to a final ASIC design — especially when monitoring requirements are still evolving.

However, from an efficiency standpoint, a dedicated monitor optimized for a specific specification offers significantly more potential for performance and resource optimization. Ultimately, the choice comes down to whether configuration happens at compile-time or at runtime — much like the classic trade-off between a compiler and an interpreter.

In this work, we have built a compiler that generates a dedicated hardware-based monitor from a particular input specification. Specifically for a stream-based language RTLola. RTLola is designed to support both online and offline monitoring — that is, monitoring live as the system runs, as well as analyzing recorded logs afterward. However, in the context of efficient hardware monitors we are interested in online real-time monitoring.

For hardware-based monitoring with RTLola, Baumeister et al. [6] have introduced a hardware architecture and have built a compiler that produces specific VHDL code from a given spec. Although the design is well thought out with a modular architecture to run asynchronous streams together, it has two major limitations.

First, the evaluation lacks pipelining, which presents a significant opportunity to improve evaluation throughput. Second, it can only produce HDL designs in VHDL, which limits its portability.

In this work, we have chosen to implement the hardware architecture in Clash, bringing in the expressive power of Haskell together with a strong type-safety that allows us to enforce some of the RTLola semantics within the HDL code. With Clash, we can generate multiple backend HDLs, so the compiler is not limited to only VHDL.

Similarly, we have extended the existing hardware architecture to enable pipeline evaluation of streams. For this, we analyze the dependency graph derived from the RTLola specification. This analysis provides an evaluation order and reveals how the evaluation can be performed in a pipelined fashion. It also determines the exact memory requirements needed to store intermediate results during pipelined evaluation. In addition, we have made several minor improvements to the hardware architecture, further enhancing the overall design.

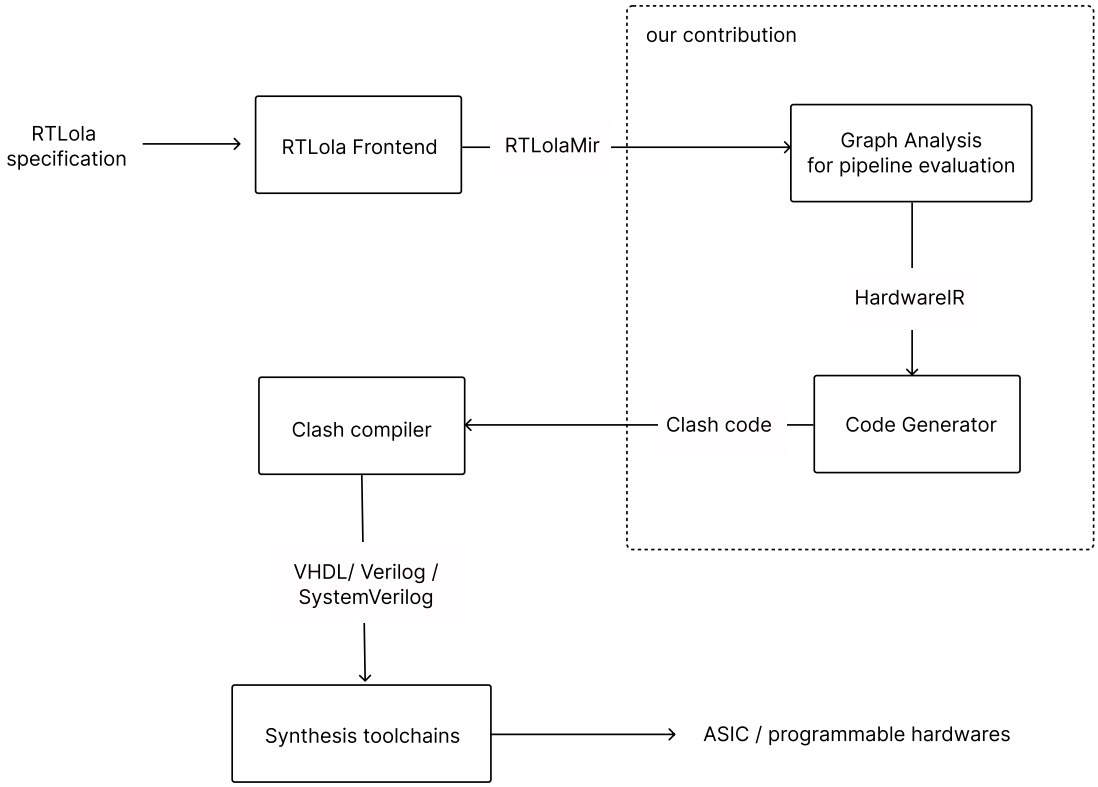


Figure 1: Hardware monitor from RTLola specification

Chapter 2

Related Work

Runtime monitoring with formal semantics initially started with temporal logic [7], [8]. Monitoring constraints can be specified using temporal logic, and a monitor can be synthesized from it [9], [10]. Later, MTL [11] and STL [12] introduced real-time properties in temporal logic. Among the synchronous languages, [13], [14], [15], Lola [1], [2] pioneered stream-based runtime verification. Copilot [16] followed Lola with its own stream-based design. With the need for input streams with varying rates in cyber-physical systems, a new class of asynchronous languages, such as RTLola [3], TeSSLa [4], and Striver [5], came into the picture.

Monitoring languages have found application in various cyber physical systems, most notably in unmanned physical systems. MTL [11] was used in NASA's Dragon Eye UAS [17] project. NASA's ICARUS project [18] used Copilot [19]. RTLola has been used [20], [21] in DLR ARTIS family of unmanned aircraft.

A monitor can be implemented as a software or a hardware. For example, Copilot [16] produces a monitor in C with bounded static memory from a DSL embedded into Haskell [22]. Scoria [23], a reactive language for IoT, and Ivory, an embedded system language, do similar things by compiling embedded Haskell into C [24]. TeSSLa [4] can produce a monitor in Scala or Rust.

Of particular interest to us is the synthesis of monitors as hardware. Several prior works have explored this direction across various temporal logics and specification languages. Dahan et al. [25] synthesized monitoring circuits directly from temporal logic specifications. P2V [26] translated sPSL specifications [27] into Verilog. BusMOP [28], [29] generated a PCI bus traffic monitor from LTL specifications for deployment on FPGAs. MBAC [30] produced hardware assertion checkers from PSL properties.

Finkbeiner et al. [31] introduced a technique to partition LTL specifications into bounded and unbounded parts, enabling efficient monitor synthesis with fewer restrictions. Jaksic et al. [32] compiled signal temporal logic (STL) specifications into FPGA-based monitors. NASA's R2U2 project [17], [33], [34] implemented monitors on FPGAs from both linear and metric temporal logic formulas.

For RTLola hardware monitor, Baumeister et al. [6] presented an approach for compiling RTLola specifications [21], [35], [36], [37] into VHDL for FPGA-based monitoring. TeSSLa [4] also enabled hardware monitoring on FPGAs [38], allowing event-processing units to be programmed via memory-mapped commands without requiring resynthesis [39],

[40]. This system has been used for detecting data races in embedded SoCs [41] and for hardware-based monitoring on the Enzian platform [42], [43].

Also more recently, in faRM-LTL [44], Benny et al. proposed a compiler that targets ASICs rather than FPGAs. Their approach uses a dedicated monitoring instruction set architecture (LM-ISA) and compiles LTL specifications directly into LM-ISA programs for efficient, ASIC-level monitoring.

On the hardware design side [45], digital circuits are traditionally developed using proprietary hardware description languages (HDLs) such as VHDL, Verilog, and SystemVerilog, along with their corresponding vendor-specific toolchains. However, the ecosystem also includes several open-source synthesis and simulation tools, such as Verilator [46], GTKWave [47], SymbiFlow [48], Yosys [49], and Icarus Verilog [50], etc.

Beyond traditional HDLs, various high-level languages and frameworks aim to generate VHDL or Verilog from more expressive abstractions. Xilinx Vivado HLS [51] and Intel's FPGA SDK for OpenCL [52] support high-level synthesis from C/C++ and OpenCL kernels, respectively. In the functional programming world, Lava [53] and Bluespec [54] were early Haskell-based HDLs. Clash [55], [56], [57], [58] has since become a solid Haskell-based HDL, enabling the use of standard Haskell syntax to generate VHDL, Verilog, or SystemVerilog. Notably, compared to other high-level synthesis tools, Clash allows high-level abstraction without losing tight low-level control.

Other high-level HDLs include Python-based frameworks such as Magma [59], MyHDL [60], and Amaranth [61]; Hardcaml [62] based on OCaml; and Scala-based HDLs like Chisel [63], SpinalHDL [64], and DFiant [65]. Also, there is TL-Verilog [66] offering a transaction-level abstraction for Verilog.

In parallel, several projects are working on high-level intermediate representations (IRs) and compiler infrastructures to facilitate hardware synthesis and accelerator design. These include MLIR [67] in LLVM/CIRCT [68], LLHD [69], and Calyx [70]. Tools like Chisel, Moore [71], and Magma are actively building on top of these modern infrastructures.

In our work, we used Clash to implement the hardware architecture for RTLola. In addition of supporting multiple HDL backends and offering an expressive hardware description language, Clash allowed us to leverage Haskell's type system to enforce certain aspects of RTLola's pacing semantics. A related effort is Hlola [72], which introduced a Haskell-embedded DSL for stream runtime verification that supports arbitrary data types. In contrast, our approach does not aim to build a full DSL, instead, we focus on partially encoding RTLola's pacing semantics directly within Haskell using its type system.

Another major contribution of our work is the introduction of pipeline evaluation for RTLola streams, an aspect not supported by the RTLola hardware compiler implemented by Baumeister et al. [6]. A related approach to pipelining was proposed by Maximilian Schwenger in his dissertation [73], where stream evaluation is stalled at runtime until all dependencies are available. This idea is conceptually similar to our `pipeline_wait` mechanism. However, our approach differs in a fundamental way: rather than stalling during evaluation, we perform a static dependency analysis at compile time to determine the necessary wait ahead of time.

This approach allows us to optimize the evaluation order by selecting one that results in shorter wait times, improving efficiency. It also enables us to predict system behavior more accurately, including how many intermediate results need to be stored and the expected evaluation throughput. Additionally, since stall handling is done at compile time, we avoid complex stalling logic in the hardware, leading to a simpler generated hardware.

Chapter 3

Background

This chapter introduces the RTLola specification language and the existing hardware architecture used for RTLola-based monitors. It then provides an overview of hardware design principles and presents the Clash hardware description language.

3.1 RTLola specification language

RTLola [3], [35] is an extension of Lola [1] designed to specify real-time constraints for reactive systems, allowing inputs to arrive at varying rates. Streams in RTLola are evaluated asynchronously, meaning they can produce results at different times depending on the availability of inputs. To allow predictable, fixed-rate outputs, RTLola introduced the concept of sliding windows with aggreation function over the window.

The dance between varying-rate and fixed-rate streams, mediated through sliding windows is the heart of RTLola and has been the focus of this work within the set of all features RTLola provides.

Let's go through all the major parts of RTLola relavent to this work.

3.1.1 Input and output streams

RTLola receives input from outside as a stream of input data and produces the results from the monitor as a stream of outputs. Such input and output streams in RTlola are specified by keyword `input` and `output` respectively. The data type of the stream is indicated by annotating it with a type. The expression to calculate the stream output is specified on the right side of the assignment symbol `:=`.

Example 3.1.1.1 (Input & output streams)

```
input distance: Float
output toofar: Bool := distance > 100.0
```

Here, `distance` is an input stream of `Float` datatype, and `toofar` is an output stream of `Bool` type. The expression `distance > 100.0` specifies how the output value is calculated: when a distance value is greater than `100.0`, the value at `toofar` will be `true`; otherwise, it will be `false`.

There is also a special type of output called `trigger` in RTLola which outputs a message when a condition is reached.

Example 3.1.1.2 (Trigger)

```
input distance: Float
output toofar := distance > 100.0
output tooclose := distance < 50.0
trigger toofar || tooclose "system out of boundary"
```

Here the message `system out of boundary` will be produced by the monitor when the condition of `toofar || tooclose` is reached.

In this work we have only focused with `input` and `output` streams ignoring the `trigger` as `trigger` can be modelled as a `Bool` type output stream.

3.1.2 Data type-system

RTLola is a strongly typed specification language. The data type-system checks the validity of types. It can also infer the data-type of a stream from the expression.

Example 3.1.2.1 (Data type-system)

```
input distance: Float
output toofar := distance > 100.0
```

Here the input stream `distance` has been annotated with a `Float` type. The output stream `toofar` gets the stream of `Float` values from `distance` and compares that withh `100.0`. From the logical comparision operator `>`, RTLola infers that the data-type of `toofar` is `Bool` so it is not necessary to explicitly speicify that in the specification.

However, if we had erronously speicified any other type to `toofar` then the type-check will fail as the explicitly speicified type doesn't match the inferred type.

Example 3.1.2.2 (Invalid data-type)

```
input distance: Float
output toofar := distance > 100
```

Here the compiler would complain that the types of operands don't match for `>` operator as the `distance` is of type `Float`, however, `100` is inferred to be of type `Integer`.

3.1.3 Event-based and Periodic streams

The varying-rate streams in RTLola get evaluated as soon as there is a related input event, hence they are also called an **event-based streams**. Whereas, the fixed-rate streams work with a fixed periodic frequency, so they get evaluated in a fixed periodic timing, hence called **periodic streams**.

Example 3.1.3.1

```
input temp: Float
input speed: Float
output toohot := temp > 80.5
output speed_delta := if toohot then -speed * 0.01 else speed * 0.01
output average_speed @1kHz :=
      speed.aggregate(over: 1s, using: average).defaults(to: 0.0)
```

Here `toohot` and `speed_delta` are event-based streams whereas `average_speed` is a periodic stream with a period of one kilo herz. The concept of `aggregate` in the expression of `average_speed` will be explained later.

3.1.4 Pacing

In Example 3.1.3.1, the `toohot` stream only depends on `temp`, so as soon as there is a new event in `temp` stream, `toohot` will be evaluated. Whereas, `speed_delta` transitively depends on `temp` via condition on `toohot` and direclty depends on `speed`. So, `speed_delta` can only be evaluated when both `temp` and `speed` have new events. However, the periodic stream `average_speed` gets evaluated every `1 ms` irrespective of the input events.

This condition of when the stream must be evaluated in RTLola is called an *evaluation pacing*. In this work, we refer to *evaluation pacing* as simply **pacing**.

So, in Example 3.1.3.1, the pacing of output streams are as follows:

- `toohot`: `@temp`
- `speed_delta`: `@(temp && speed)`
- `average_speed`: `@1kHz`

which indicates that `toohot` must be evaluated when there is an event *at* `temp`, `speed_delta` must be evaluated when there is an event *at* `temp` and *at* `speed`, and so on.

3.1.5 Pacing type-system

Just as RTLola performs type-checking on data types, it also includes a type system for pacings. This system helps prevent a wide range of timing-related bugs in stream evaluations.

Example 3.1.5.1 (Invalid event-based pacing)

```
input x: Int
input y: Int
output a @x := x * 2
output b @y := a + y
```

Here the output `b` has been explicitly annotated with pacing `@y`. However the expression depends on `a` and `y` where `a` has pacing of `@x`. As the streams are asynchronous, when there is a new value in `y` there might not be a new value in `x` as well. This means the stream `b` can only be evaluated when there is a new value in both `x` and `y` i.e the pacing of `@(x && y)`. Hence, the RTLola will reject this specification.

Example 3.1.5.2 (Invalid periodic pacing)

```
input x: Int
input y: Int
output a @1Hz := x.aggregate(over: 1s, using: sum)
output b @2Hz := y.aggregate(over: 1s, using: sum)
output c @2Hz := a - b
```

Here the output `c` works on the values from stream `a` and `b`. The output `a` has a frequency `1Hz` so is evaluated every `1 second`. Whereas, the output `b` is evaluated every `0.5 seconds`. As the result from `a` comes less frequently, the stream `c` can't be evaluated every `0.5 seconds` as indicated by the `@2Hz` pacing. Hence, the RTLola type-checker rejects the specification. However, RTLola would have accepted the pacing of `@1Hz`, `@0.5Hz`, etc. in output stream `c` as the both `a` and `b` would have values in those cases.

In the absence of explicit pacing, the RTLola type-system tries to infer the pacing. In the example, if we hadn't provided any pacing for `c`. Then the RTLola would have inferred the pacing of `@1Hz` which is the most frequent pacing it can have in this particular case.

3.1.6 Offset access

In RTLola, past and future values of streams can be accessed via offsets. However, as of now, the future access is not fully supported in RTLola.

Example 3.1.6.1 (Past offset)

```
input distance: Int
output getting_closer :=
    distance < distance.offset(by: -1).defaults(to: distance)
```

Here the output `getting_closer` is calculated by comparing the current value of `distance` with the most recent past value. The `distance.offset(by: -1).defaults(to: distance)` corresponds to the access of past value. The `offset` value is relative to the present. Here `-1` means the last value, i.e the value `1` step before. `-2` would mean the value `2` steps before, and so on. As the stream could have been just started, there might not be any past value of the stream. In such cases, `defaults(to: distance)` specifies the default of current value of `distance` to choose.

3.1.7 Hold access

RTLola provides a way to access asynchronous value from another stream via a `hold` access.

Example 3.1.7.1 (Hold access)

```
input x : Int
input y : Int
output a @x := x + 1
output b @x := a + y.hold(or: 0)
```

In this example, both streams `a` and `b` are evaluated whenever a new value arrives on the input `x`, as specified by the pacing annotation `@x`. However, the expression defining `b` also depends on the input `y`, which may not have a new value at the same time, since `x` and `y` are asynchronous inputs. To handle this, RTLola provides the `hold` access, which retrieves the most recent value of a stream. Using `y.hold` in the expression ensures that the latest available value of `y` is used—if `y` updates at the same time as `x`, the new value is used; otherwise, the last known value is used. If the stream `y` has just started and has no history, the default value specified by `or: 0` will be used instead.

Hold access works in similar way for periodic streams as well.

Example 3.1.7.2 (Hold access in periodic streams)

```
input x : Int
output a @2s := x.hold(or: 0) + 1
output b @1s := a.hold(or: 0)
```

Here, when `a` is evaluated i.e every `2s`, the stream `x` might not have the value at that instant. So, we use the lastest value of `x` using the `hold()` access. Similarly, the pacing of output `a` and `b` is different. Hence, the use of `hold` access in output `b` to access the value from the slower stream `a`.

3.1.8 Sliding window aggregation

Sliding window aggregation is a major feature of RTLola. An aggregation function combines all the values within a time window in the stream to produce a predictable, fixed-rate stream from an otherwise unpredictable, asynchronous one.

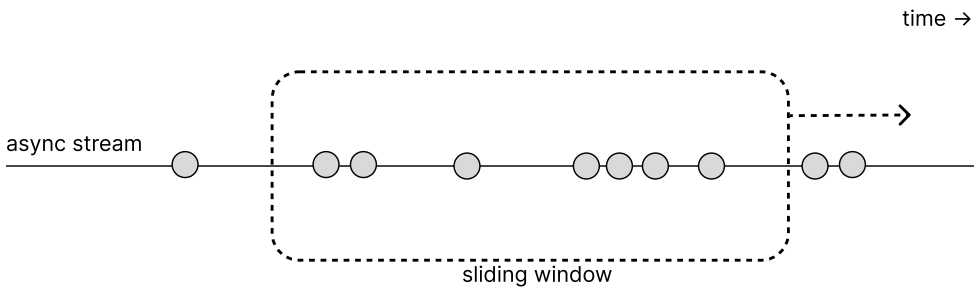


Figure 2: Snapshot of a sliding window

Example 3.1.8.1 (Sliding window aggreation)

```
input sold_price : Float
output items_sold @1s := sold_price.aggregate(over: 10s, using: count)
output revenue @1s := sold_price.aggregate(over: 10s, using: sum)
```

In this imagined sales monitoring system, sales can occur at any time and with any frequency. However, statistics such as total `revenue` and the number of `items_sold` over the past ten seconds are computed and reported every second. These sliding windows have a span of `10s`, and the aggregation functions used are `sum` for revenue and `count` for items_sold.

3.1.9 Efficient evaluation of sliding windows with bounded memory

The number of events within a span of a sliding window is not predetermied. If we need to remember the events to aggregate, we can't be sure how much memory our monitor would need. This unbounded memory is a big challenge as we are never sure when the system will crash being out of memory. However, if we can immediately collect a partial aggregation from the event and don't have to remember the event any longer then we won't have this problem. RTLola [3] uses this observation to efficiently perform sliding window aggreation with *homomorphic* aggreation functions such as `max`, `min`, `sum`, etc.

As the sliding window advances, the part of the past events goes outside the window and the new events fall within the window. So the same events are repeatedly used in aggregation. With the memoization of partials we don't have to recalculate the events again and again making evaluation very efficient.

According to the pacing and span, RTLola quantizes the time into a fixed step size. The sliding window forwards in time in this quantized step. For each step, it keeps a bucket of partial aggregated values. So, an aggregation window can have a number of partially aggregated buckets. At each slide the earliest bucket will go out of range along with its partially aggregated values and the new fresh bucket on the other end will start. When there is an event, the event is immediately aggregated into the latest bucket. When it is time to output the aggregated result the buckets are aggregated together to produce a single value.

3.1.10 Advanced features of RTLola

RTLola has other features such as parametrization of streams and stream life-cycle. These features are very powerful interms of expressive power it gives to the specification. However, in our work focusing on hardware monitors, they are not in our scope.

3.1.11 Dependency graph

The dependencies between streams in RTLola produces a graph structure.

Definition 3.1.11.1 (Dependency Graph): Let $S_i = \{i_1, i_2, ..., i_p\}$ be the set of input streams, $S_o = \{o_1, o_2, ..., o_q\}$ be the set of output streams, and $S_w = \{w_1, w_2, ..., w_r\}$ be the set of sliding windows s.t $S_w = \bigcup_{j=1}^{q} w_{o_j}$ where w_{o_j} is the set of all sliding windows aggregates associated with output stream o_j.

The **dependency graph** of an RTLola specification is a directed multi-graph $G = \langle V, E \rangle$ where $V = S_i \cup S_o \cup S_w$ and $e = \langle v_i, v_j, w \rangle \in E$ which corresponds to the access of data from v_i by v_j, with w being the offset weight in the offset-access and zero in all other accesses. [35]

3.1.12 Well-formed specification

RTlola performs well-formedness analysis to avoid specifications with semantically invalid cycles in the dependency graph.

Example 3.1.12.1 (Invalid cycle)

```
input x : Int
output a := x + b
output b := x + a
```

Here there is a cycle $a \rightarrow b \rightarrow a$. They depend on each other creating a data-race with infinite feedback with no clear order. Such specifications are not wellformed. To avoid such cycles, RTLola checks all the cycles in the graph and enforces that the accumuated weight of the edges are not zero [3].

Definition 3.1.12.1 (Well-formedness): A RTLola specification is *well-formed* iff there is no cycle in the dependency graph with the accumulated weight of zero. [1]

3.1.13 Evaluation order

The RTLola streams must be evaluated in the order corresponding to the dependency between them. Following the definition from Schwenger [74] and Baumeister [75], the evaluation order can be defined as:

Definition 3.1.13.1 (Evaluation order): The *evaluation order* ($\rightarrow$) is a partial order on streams, reflecting the structure of the dependency graph $G =< V, E >$. The evaluation order is the transitive closure of a relation satisfying following rules [75]:
1. $\forall i, j : \text{input\_stream}_i \rightarrow \text{output\_stream}_j$
2. $\forall e = \langle u, v, w \rangle \in E$ with $w = 0$, $u \neq v$ if $u \in S_i$ and $v \in S_j$, then $u \rightarrow v$

3.1.14 Cycles in RTLola

RTLola specificationscan have cycles. To obtain a correct evaluation order, it is important to understand various kinds of cycles that can exist between depedencies in RTLola.

Looking categorically, cycles in RTLola can occur in 3 different ways:

3.1.14.1 Cycles in event-based streams via negative offsets

Wellformedness analysis (Section 3.1.12) implies that the cycle in RTLola must occur via an `offset`. Currently, RTLola only supports -ve offsets so the cycle in event-based streams must occur via it.

Example 3.1.14.1.1 (Cycle in event-based streams)

```
input x: Int
output a := x + c.offset(by: -1).defaults(to: 1)
output b := a + 1
output c := b + 1
```

3.1.14.2 Cycles in periodic streams via negative offsets

Similary, the cycle can exist in the periodic streams via -ve offsets.

Example 3.1.14.2.1 (Cycle in periodic streams)

```
output a @1kHz := a.offset(by: -1).defaults(to: 0) + 1
                  c.offset(by: -1).defaults(to: 1)
output b := a + 1
output c := b + 1
```

3.1.14.3 Cycle across event-based and periodic streams via holds

Apart from `offset` based cycles, there can also be a cycle across event-based and periodic streams due to `hold` access. The wellformedness analysis (Section 3.1.12) has been loosen in this case for practicality.

a. RTLola's event-based before periodic rule

The `hold` access in RTLola can lead to a cycle without a clear order of evaluation.

Example a..1 (Valid cycle without clear order)

```
input x: Int
output a := x + b.hold(or: 4)
output b @1Hz := a.hold(or: 0)
```

In this specification, there is a cycle $a \to b \to a$ without a clear order. RTLola exempts such specifications from the wellformedness analysis because they have a valid usecase in practice.

Example a..2 (Practical use-case)

```
input x: Int
output a := if sum_so_far.hold(or: 0) > 10
            then x
            else x + sum_so_far.hold(or: 0)
output sum_so_far @1kHz := a.aggregate(over: 1s, using: sum)
```

If we need to choose a value based on an aggregate, such as in the example Example a..2, we must allow for such cyclic dependencies. However, evaluating such specifications requires enforcing a specific evaluation order. RTLola resolves this by always evaluating event-based nodes before periodic nodes. In the example above, the evaluation order would be: $x \to a \to$ sum_so_far.

3.2 Existing hardware architecture for RTLola

Baumeister et. al [75], [6] introduced a hardware architecture for RTLola with implementation in VHDL. The architecture has three main components that are namely, high-level controller, first-in first-out queue, and the low-level controller. With this architecture the timing/pacing of streams is seperated from the actual evaluation, producing components that can work asynchronously of each other with their own clocks.

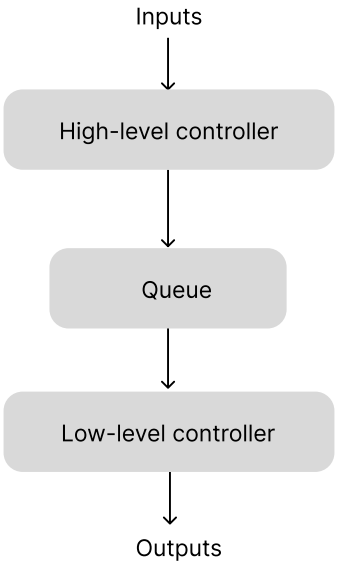


Figure 3: Existing general architecture

3.2.1 High-level controller (HLC)

The High-Level Controller (HLC) is responsible for receiving inputs to the system and generating activation conditions for streams, known as pacing (Section 3.1.4). Since RTLola supports two types of streams, event-based and periodic (Section 3.1.3), the HLC must manage both types appropriately to ensure correct evaluation.

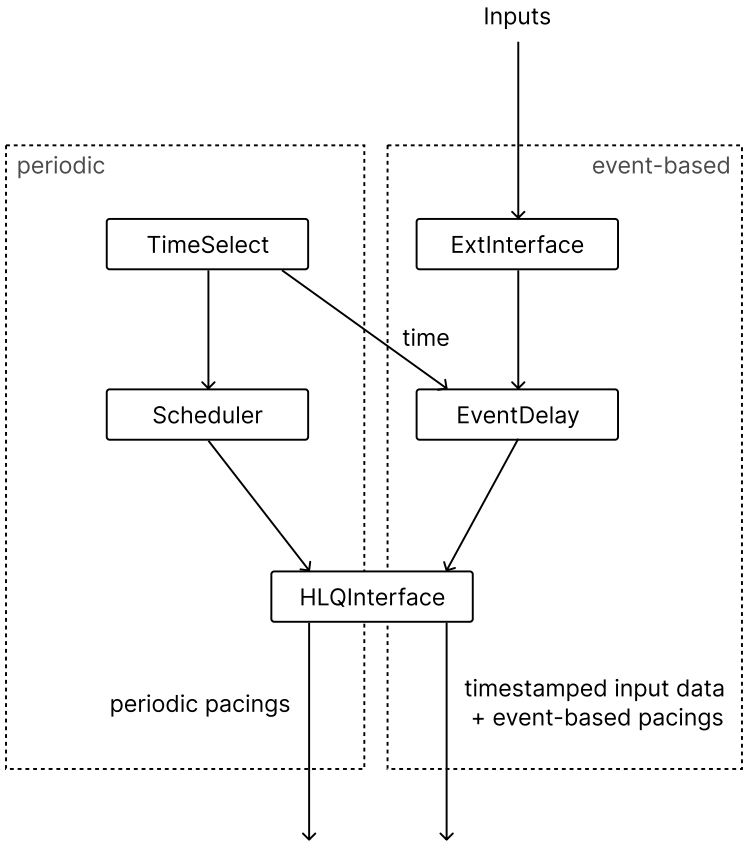


Figure 4: Structure of high-level controller

Figure 4 illustrates the internal architecture of the high-level controller (HLC). The HLC operates in a pipelined manner, synchronized by a shared clock.

Its periodic component consists of two main parts: *TimeSelect* and *Scheduler*. The *TimeSelect* computes the current system time based on the clock frequency and the number of clock cycles elapsed since the start. The *Scheduler* then uses this system time to generate pacing signals for periodic streams at the correct moments.

The event-based part of HLC consists of *ExtInterface* and *EventDelay*. *ExtInterface* interfaces outside to receive the inputs. The inputs are annotated with the system time by *EventDelay*.

The existing monitor supports both online and offline monitoring. In offline mode, inputs are provided along with their timestamps, so they must be delayed until their actual activation time. This delay is handled by the *EventDelay*, hence the name. The delay is not necessary during online monitoring.

Remark 3.2.1.1 (Input is annotated with time). *The HLC annotates inputs with their corresponding timestamps to determine where each input falls within the sliding window span during evaluation.*

The *HLQInterface* then interfaces with the first-in-first-out queue to forward the pacing informations along with the input data. It submits the results from *EventDelay* and *Scheduler* alternately to the queue.

Remark 3.2.1.2 (Alternate evaluation of event-based and periodic streams). *According to RTLola's semantics, an input event that arrives in the same cycle as a periodic deadline must be included in the evaluation. This requires evaluating event-based streams before periodic ones. Additionally, cyclic dependencies between the two types of streams can introduce race conditions, as described in Section 3.1.14.3. Evaluating event-based before periodic also introduces a fixed order to break such race-conditions. Consequently, the HLQInterface cannot submit results from EventDelay and Scheduler simultaneously. Instead, it submits them alternately in that order. Since both stream types can be activated in a single clock cycle, the alternate evaluation requires the HLQInterface to have a faster clock running at twice the speed to prevent the information loss.*

3.2.2 Queue

The High-Level Controller (HLC) receives inputs and calculates pacing signals at a much higher rate than the Low-Level Controller (LLC) can process them. This creates a natural need for a buffer to interface between the two. Since inputs and pacing signals must be processed in the order they arrive, this buffer is implemented as a first-in, first-out (FIFO) queue. Additionally, the queue helps absorb sudden bursts of input, provided there is sufficient memory available.

3.2.3 Low-level controller (LLC)

Low-level controller is responsible for the actual evaluation of streams in RTLola. The evaluation must happen according to the evaluation order (Section 3.1.13) based on the interdependencies between streams. LLC evaluates streams step by step in the evaluation order without pipeline, hence it is considerably slower than HLC. Only after fully evaluting all the streams can it pop the event from the queue to do another cycle of evaluation.

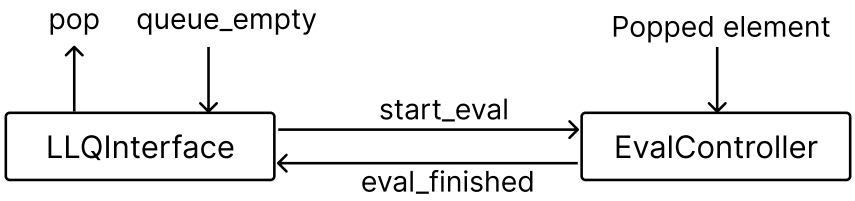


Figure 5: Structure of low-level controller

Figure 5 shows the inner architecture of LLC which consists of two parts: *LLQInterface* and *EvalController*. *LLQInterface* is responsible for interfacing with the queue and popping the elements from the queue. On valid event from a queue, *EvalController* starts evaluation of streams in the correct order.

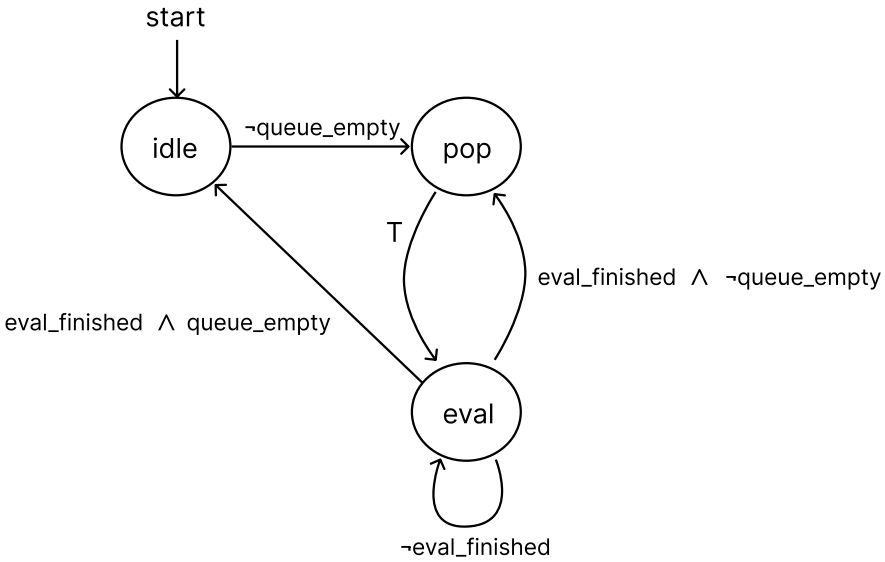


Figure 6: LLQInterface state machine

Figure 6 shows the states of *LLQInterface*. *LLQInterface* pops the info from the queue when its ready. Once it has a valid data it starts the evaluation. It stays in the *eval* state until *EvalController* has completely finished the evaluation.

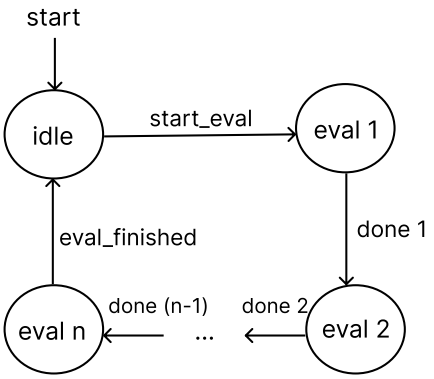


Figure 7: EvalController state machine

Figure 7 illustrates the states of the *EvalController*. Each state corresponds to a specific position in the evaluation order. At each evaluation state, only the streams in that position in the order are evaluated. The evaluation process completes once all streams have been processed in this manner. Afterward, the *LLQInterface* becomes ready to pop and handle new incoming data. The monitor outputs the final results only after all evaluation steps are completed.

Remark 3.2.3.1 (LLC keeps track of time for sliding windows). *LLC keeps track of time until which it must slide the sliding windows. It compares the system time with this deadline to know if it is time to slide.*

Remark 3.2.3.2 (Number of eval steps in LLC is not fixed). *The evaluation finished signal is generated when all active streams are evaluated according to the evaluation order. Since event-based and periodic streams may have dependency chains of different lengths, the number of evaluation steps in the alternating evaluation of event-based streams can differ from that of periodic streams.*

3.3 Hardware design with Clash

3.3.1 Hardware design process

A custom hardware can be built on top of a reprogrammable hardware such as FPGAs or as its own specific chip (ASIC). The design of hardware usually starts with a hardware description in a hardware description languages (HDL) such as Verilog, VHDL or SystemVerilog. The HDL design goes through verification phase including software simulation. From the finalized design, the synthesis process starts. The HDL code destined to an FPGA goes through series of tranformation from the intermediate netlist form with optimzations based on the physical constraints and resources of the actual end device. Similary for ASIC, the series of synthesis process of transformation and optimzation produces schema that will be use to fabricate the chip.

3.3.2 Clash

Clash [55], [56], [58] is a hardware description language built on top of the functional programming language Haskell [22]. It follows the syntax and symantics of Haskell to allow for a high-level expressive description of hardware. Clash compiler can transform the descritpion of hardware in Clash into traditional HDLs like VHDL, Verilog and SystemVerilog [57]. The existing toolchains for traditional HDLs can then be used to synthesize a hardware.

In the following sections, we explore Clash and Haskell in the context of this work. For a general introduction to functional programming, Haskell and Clash, readers are advised to follow the resources in references [22], [58], [76], [77], [78], [79], [80].

3.3.2.1 High-level abstraction & expressiveness

Clash provides high-level abstractions to design hardware circuits. To get a feel for it, consider the example below of FIR circuit in Clash taken from the Clash website (https://clash-lang.org/).

Example 3.3.2.1.1 (FIR filter in Clash)

```
fir coeffs x = dotp coeffs (window x)
  where
    dotp as bs = sum (zipWith (*) as bs)


fir3int = fir (3 :> 4 :> 5 :> Nil)
fir4float = fir (3.5 :> 4.2 :> 3.0 :> 6.1 :> Nil)
```

FIR filter is a digital signal processing filter that performs discrete convolution on a signal to produce a filtered signal. To obtain a filtered signal, a sliding window slides through the input signal where each of the values in the window is weighted by the coefficient of a filter and added together i.e a dot product operation.

In Example 3.3.2.1.1, the FIR filter is defined as a function that takes a vector of coefficients and an input signal, and produces an output by computing the dot product between the coefficients and a sliding window over the input signal. The `zipWith` function in Clash is a high-level function that takes two vectors of equal length and applies a given function with element-wise pair, producing a combined new vector. The concrete filters `fir3int` and `fir4float` are partial applications of this FIR function, where the coefficients are predefined. Since `zipWith` requires vectors of the same length, Clash infers the size of the sliding window `(window x)` to match the length of the coefficients vector. Similarly, the data type of the signal is inferred from the type of the provided coefficients.

The example shows how Clash provides a very expressive way to build circuits with modular parametric design benefiting readability, maintainability and code reuse.

3.3.2.2 Finite hardware from a lazy language

Haskell is a lazy functional language [77], meaning expressions are evaluated only when their results are needed. Unevaluated expressions are represented as thunks, which describe how a value can be computed when required. Once evaluated, a thunk is replaced by the computed value and shared across all references, thanks to pure functions in Haskell. This laziness makes Haskell highly expressive, enabling features like infinite lists and recursive definitions without significant performance issues.

However, Haskell's lazy evaluation model contrasts sharply with the eager nature of hardware circuits. In hardware design, the number and size of components must be known ahead of time. To bridge this gap, Clash restricts Haskell to a synthesizable subset - ensuring that the resulting description maps cleanly to a finite, static hardware circuit.

Related type classes in Clash, such as `NFDataX` and `KnownNat`, are frequently used in this work. The `NFDataX` type class ensures that a value can be fully evaluated, even if it contains unknown values (i.e., `DataX`). The `KnownNat` type class guarantees that any natural number used in a parameterized function is known at compile time, which is essential for hardware synthesis.

3.3.2.3 Sequential circuit design

Sequential circuits are inherently stateful, storing past values using memory elements such as latches or flip-flops. The transitions between states must be synchronized, which is typically achieved using a shared clock signal.

In Clash, this concept is modeled using the `Signal` type. A sequential circuit operates on values of type `Signal dom a`, representing an infinite stream of values of type `a`, sampled at each tick of the clock. The `dom` parameter refers to a specific clock domain, defined by the `ClockDomain`, which specifies properties such as the clock frequency, edge triggering (positive or negative), and reset behavior.

The `register` function in Clash is the primary way to store state. It corresponds to a latch that is synchronized by the global clock. The expression `register initial sig` represents the latching of the signal `sig` at each clock tick, starting with an initial value `initial`. Thinking in terms of an infinite stream sampled at each clock, register delays the input signal by one clock cycle and begins with the initial value. When a global reset signal is used, the register is re-initialized to the initial value.

Example 3.3.2.3.1 (Sequential circuit in Clash: 4-bit counter)

```
module FourBitCounter where

import Clash.Prelude

counter :: HiddenClockResetEnable dom => Signal dom (Unsigned 4)
counter = register 0 (counter + 1)

topEntity :: Clock System -> Reset System -> Enable System -> Signal
System (Unsigned 4)
topEntity = exposeClockResetEnable counter
```

Example 3.3.2.3.1 is a four-bit counter implemented in Clash. The code starts by defining a module and importing the standand library. Then the `counter` is defined as a `Signal` of 4-bit unsigned number. The statement `counter = register 0 (counter + 1)` indicates that the `counter` starts with `0` and at each clock trigger the `counter` increments by `1`. Here `counter + 1` is a signal built from the last value of `counter`. Register needs a clock signal to work. The `HiddenClockResetEnable` type constraint in `counter` function ensures that the `Domain dom` implicitly has a clock, reset and an enable signal which is exposed to the `counter` at `topEntity`.

3.3.2.4 Strong static type-system

Clash inherits the strong static type-system from Haskell which lets it discover various issues in the circuit at compile time, such as incorrect clocking, uninitialized registers, etc.

To better understand this, let's examine a few example scenarios:

Example 3.3.2.4.1 (Mixing of clock domains)

```
module TypeSystem where

import Clash.Prelude

createDomain vSystem{vName="DomainA", vPeriod=1000} -- 1MHz
createDomain vSystem{vName="DomainB", vPeriod=2000} -- 500kHz

adder :: (HiddenClockResetEnable dom, Num a)
    => Signal dom a
    -> Signal dom a
    -> Signal dom a
adder x y = x + y

a :: Signal DomainA Int
a = pure 1

b :: Signal DomainB Int
b = pure 2
```

In Example 3.3.2.4.1, two clock domains — `DomainA` and `DomainB` — are defined based on the `System` domain. The function `adder` performs the addition of two signals, each carrying numeric data, as indicated by the `Num` type constraint. Both signals must belong to the same clock domain `dom`, ensuring that they are synchronized during computation.

```
c = adder a b
-- compile error: Couldn't match type 'DomainB' with 'DomainA
```

In this example, the `adder` function is invoked with signals from different clock domains — one driven by a `1 MHz` clock and the other by a slower `500 kHz` clock. In a real hardware circuit, this mismatch could lead to subtle and hard-to-detect bugs. However, thanks to Clash's strong type system, such issues are caught at compile time, preventing incorrect cross-domain connections.

```
c = adder (pure 1 :: Signal DomainA (Unsigned 2)) b
-- compile error: Couldn't match type 'Int' with 'Unsigned 2'
```

In this case, the `adder` function was called with signals carrying incompatible data types, which was detected at compile time.

Example 3.3.2.4.2 (Uninitialized register)

```
fn :: HiddenClockResetEnable dom => Signal dom Int -> Signal dom Int
fn input = register input
-- compile error: 'register' is applied to too few arguments
```

If a register is not properly initialized, it may start with a random value after a reset, potentially introducing subtle bugs. Clash prevents this by requiring a default (reset) value

to be explicitly provided as an argument. In Example 3.3.2.4.2, the register was used without specifying a reset value, which was found by the Clash type system, as the register expects an initial value. A correct usage would be something like `register 0 input`, where `0` serves as the reset value.

3.3.2.5 Recursion

Clash is a structural hardware description language where each function is circuit component and every call of a function is an instantiation of the component. In a structural view, a general recursive function can lead to an infinitely nested component instantiation which is not possible to synthesize into a finite hardware. So Clash has a very limited support for recursion [79]. However, Clash offers range of higher-order functions which provides a workaround for recursion in most cases.

3.3.2.6 High-order functions

Like Haskell, Clash supports higher-order functions i.e the parameters and the return type of a function in Clash can be a function itself.

Example 3.3.2.6.1 (Higher-order functions)

```
combiner :: (Num a) => (a -> a) -> (a -> a) -> a -> a
combiner f1 f2 x = (f1 x) + (f2 x)

increment :: (Num a) => a -> a -> a
increment step x = step + x

result = combiner (increment 1) (increment 2) 1
```

In Example 3.3.2.6.1, `combiner` is defined as a function that takes in two functions that work on `Num`. The provided functions are individually applied to the value and added together to produce a result. The `result` is produced by combining the results from the individual `increment` functions where `increment 1`, `increment 2` are functions produced from partial application.

3.3.2.7 Pattern matching

Haskell pattern matching is supported in Clash. Pattern matching allows to deconstruct the values to choose behaviour based on the structure of the data.

Example 3.3.2.7.1 (Pattern matching)

```
sumFirstTwo Nil = Nothing
sumFirstTwo (x :> Nil) = Nothing
sumFirstTwo (x :> y :> xs) = Just (x + y)

myVecSum v = case sumFirstTwo v of
               Just x -> x
               Nothing -> 0
```

Example 3.3.2.7.1 has a function `sumFirstTwo` which returns a monadic `Maybe` value depending on the size of the `Vec`. It pattern matches and returns with `Nothing` when the

vector doesn't have enough elements. For example, the `x :> Nil` pattern matches on the shape of the vector with only one element.

The `myVecSum` function again pattern matches on the result from the `sumFirstTwo` function. It returns with the inner value when the value is there; otherwise, it returns `0`.

Remark 3.3.2.7.1 (Monad). *Monad in functional programming is a container for a value with extra behaviour attached that allows to elagently chain operations over it. For example, `Maybe` is aware if the value is there. Chaining the operations over it as follows, we get a composition of operations that could end at any point due to failure but keeps going as long as there is a valid data.*

```haskell
division :: Float -> Float -> Maybe Float
division x y = if y == 0 then Nothing else Just (x / y)

squareRoot :: Float -> Maybe Float
squareRoot x = if x < 0 then Nothing else Just $ sqrt x

-- chaining of division and squareRoot
calculate :: Float -> Float -> Maybe Float
calculate x y = division x y >>= squareRoot
```

3.3.2.8 Algebraic data type

In Haskell, an algebraic data type is a composite type made up of other types. It can be a *sum* of types i.e variants or the *product* of types i.e combination.

Example 3.3.2.8.1 (Sum type)

```haskell
-- Algabriac data type 'State'
data State = Ready | Processing | Error Int | Success -- variant/sum type

resultReady :: State -> Bool
resultReady state = case state of
                        Success -> True
                        _ -> False

getErrorCode :: State -> Maybe Int
getErrorCode state = case state of
                        Error x -> Just x
                        _ -> Nothing
```

In Example 3.3.2.8.1 `State` is an algebraic data type that can be one of the variants. The `resultReady` and `getErrorCode` functions show how the variants can be pattern matched.

Example 3.3.2.8.2 (Product type - tuple)

```haskell
data State = Ready | Processing | Success deriving (Generic, NFDataX)
type CyclesSpentInState = (State, Int) -- tuple of types
```

In Example 3.3.2.8.2, the types `State` and `Int` are combined into a tuple. With `type` keyword, Haskell treats `CyclesSpentInState` as an alias for the tuple and not an actual new type.

Example 3.3.2.8.3 (Product type - record)

```
data NewData = NewData {
    value :: Int,
    valid :: Bool
} deriving (Generic, NFDataX)
```

Example 3.3.2.8.3 shows a record type called `NewData` defined as a combination of `Int` and `Bool` types. Record types allow to give names to the fields and are very flexible in creating a readable and maintainable code.

Remark 3.3.2.8.1 (Deriving NFDataX). *Derivation of NFDataX is important in the type of data in Clash inorder for Clash to know that it can fully evaluate the data for synthesis.*

3.3.2.9 Signal is an Applicative Functor

Clash's `Signal` type is an applicative functor [80].

Remark 3.3.2.9.1 (Functor). *A functor in Haskell is a container with a value inside that allows the application of function from outside with `fmap` without changing the structure of the container.*

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Remark 3.3.2.9.2 (Applicative). *An applicative in Haskell is a container that can wrap a function inside of it and allows the application of the wrapped function into a wrapped value.*

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Example 3.3.2.9.1 (Signal is a Functor)

```
mult3 :: HiddenClockResetEnable dom  => Signal dom Int -> Signal dom Int
mult3 = fmap (*3)

add2 :: HiddenClockResetEnable dom => Signal dom Int -> Signal dom Int
add2 x = (+2) <$> x
-- <$> is an alias for `fmap`
```

Example 3.3.2.9.2 (Signal is an Applicative)

```
fs = fromList [(+1), (*2), (`div` 2)] :: Signal dom (Int -> Int)
xs = fromList [1, 2, 4] :: Signal dom Int

zs :: Signal dom Int
zs = fs <*> xs
```

```
clashi> sampleN @System 3 zs
[2,4,2]
```

In Example 3.3.2.9.2, we have an applicative `Signal` with functions wrapped inside of it in `fs`. The `xs` has the values wrapped inside of it. The result `zs` is a `Signal` with the wrapped function applied to the wrapped value.

The second block of code shows the actual result when running the code in the Clash interpreter with a system clock. Here `@System` indicates `System` clock domain with sampling of `3` values from the infinite signal `zs`.

Example 3.3.2.9.3 (Signal is an Applicative Functor)

```
mult :: HiddenClockResetEnable dom
    => Signal dom Int
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom Int
mult x y z = mult3 <$> x <*> y <*> z
    where
      mult3 :: Int -> Int -> Int -> Int
      mult3 a b c = a * b * c
```

```
clashi> x = fromList [1,2,3] :: Signal dom Int
clashi> y = fromList [2,3,4] :: Signal dom Int
clashi> z = fromList [4,5,6] :: Signal dom Int
clashi> sampleN @System 3 (mult x y z)
[8,30,72]
```

In Example 3.3.2.9.3, we have a function `mult3` defined on `Int` data types. The `mult3 <$> x` creates a partial function wrapped inside which can be applied to the wrapped data using `<*>` in curried fashion.

Remark 3.3.2.9.3 (Currying). *Currying is a technique of transforming function with multiple arguments into a sequence of partial functions that take one argument.*

So a function:

```
add :: Int -> Int -> Int
add x y = x + y
```

When applied partially produces a function like this:

```
add :: Int -> (Int -> Int)
add x = \y -> x + y
```

Let's consider a practical example that shows how these concepts can be applied.

Example 3.3.2.9.4 (Practical example)

```
data State = Ready | Processing | Success deriving (Generic, NFDataX)
type CyclesSpentInState = (State, Int)

nextState :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom CyclesSpentInState
nextState canTansition = output
    where
        output = register (Ready, 0) nextOutput
        (curState, cyclesSpent) = unbundle output

        nextOutput = bundle (stateSignal, cyclesSignal)
        stateSignal = stateFx <$> canTansition <*> curState
        cyclesSignal = cyclesFx <$> canTansition <*> cyclesSpent

        stateFx :: Bool -> State -> State
        stateFx canTransit state = case (canTransit, state) of
                    (False, _) -> state
                    (True, Ready) -> Processing
                    (True, Processing) -> Success
                    (True, Success) -> Ready

        cyclesFx :: Bool -> Int -> Int
        cyclesFx canTransit cycles = case canTransit of
                    False -> cycles + 1
                    True -> 0
```

In Example 3.3.2.9.4, we have a function `nextState` that transitions machine into a different state based on the active `canTransition` signal. The machine also counts the number of cycles it stayed in any particular state. The algebraic data types `State` and `CyclesSpentInState` provide the necessary data structure for this. The function `stateFx`

and `cyclesFx` handle the transition logic but they are defined on the normal types. The normal functions are raised to work with `Signal` with the help of `<$>` and `<*>`.

Remark 3.3.2.9.4 (Bundle and Unbundle functions). *The `bundle` function in Clash converts tuple of signals into a signal of tuples and `unbundle` function converts a signal of tuples into a tuple of signals.*

```
bundle :: (Signal dom a, Signal dom b) -> Signal dom (a, b)
unbundle :: Signal dom (a, b) -> (Signal dom a, Signal dom b)
```

3.3.2.10 High-level functions in Clash

Clash provides many high-level functions. Some of them we have already seen before, like `zipwith`, `register`, `bundle`, `unbunndle`, etc. There are too many to explore exhaustively, but let us examine a few more with examples to gain a better understanding.

Example 3.3.2.10.1 (Vec, mealy, foldl, head)

```
data State = Ready | Processing | Done deriving (Generic, NFDataX)
type Input = Vec 3 Float
type Output = Float

transitionFx :: State -> Input -> (State, Output)
transitionFx state input = (newState, output)
  where
    newState = case state of
      Ready -> Processing
      Processing -> Done
      Done -> Ready
    output = foldl (/) ((head input) * (head input)) input

mealyMachine :: HiddenClockResetEnable dom =>
                Signal dom Input -> Signal dom Output
mealyMachine = mealy transitionFx Ready
```

In Example 3.3.2.10.1, `mealyMachine` is built using the `mealy` function in Clash. The `mealy` function takes a `transitionFx`, which processes the current `State` and `Input` to produce a new state along with the corresponding output. The `mealy` function wraps the normal transition function along with the initial state i.e `Ready` in this example, to produce a mealy machine from the signal of input to the signal of output.

The `Input` has been defined as the `Vec` (Vector) of `3` `Float` items. The `head` function gives the first item from the vector. The `foldl` function takes a binary function and an initial value to reduce the `Vec` from left to right.

Example 3.3.2.10.2 (mux, all, .&&.)

```
type IsValid = Bool

machine :: HiddenClockResetEnable dom =>
           Signal dom (IsValid, Input) -> Signal dom (IsValid, Output)
machine hasInput = register (False, 0) hasOutput
  where
    hasOutput = bundle (outputIsValid, output)
    outputIsValid = (all (/= 0) <$> input) .&&. inputIsValid
    output = mux inputIsValid
                 (mealyMachine (reverse <$> input))
                 (pure 0.0)
    (inputIsValid, input) = unbundle hasInput
```

Example 3.3.2.10.2 continues on the Example 3.3.2.10.1. Here `mux` function has been used to choose the output signal based on a condition. The condition is a `Signal` of `Bool`. The `.&&.` function performs the logical *AND* operation on the `Signal` of `Bool`. The `all` function in this example determines whether all elements of the *Vec* satisfy the condition that each item is non zero.

Example 3.3.2.10.3 (Testing in Clash Interpreter & repeat function)

```
clashi> let input = fromList [(repeat 0.0),(repeat 0),(1:>2:>4:>Nil),
(repeat 0),(1:>3:>9:>Nil)]
clashi> let isValid = fromList [False, False, True, False, True]
clashi> let inputSignal = bundle (isValid, input)
clashi> sampleN @System 6 (machine inputSignal)
[(False,0.0),(False,0.0),(False,0.0),(True,2.0),(False,0.0),(True,3.0)]
```

In Example 3.3.2.10.3, we test the design in Clash interpreter by running against the test input signals. Here `repeat 0` produces the `Vec` where each item is `0`. The size of the `Vec` is inferred from the context. Similary from the first data type of `0.0` in `repeat 0.0` the data of the `Signal` is also inferred to be a `Float` type.

3.4 Remarks

While we haven't covered every aspect of RTLola, Haskell and Clash, the reader should now have sufficient understanding to follow this work. For further exploration, readers are advised to follow the resources mentioned in the references.

Chapter 4

New Hardware Architecture With Pipeline Evaluation

This chapter presents a new hardware architecture for RTLola monitors, built upon the existing design, that enables pipelined stream evaluation. It discusses the limitations of the original architecture and proposes improvements to address them. Additionally, it explores the limits of the new design.

4.1 Limitations of the existing architecture

The existing hardware architecture for RTLola monitor [6] has been explained in detail in Section 3.2. The design has two major limitations and some design choices that can be improved.

4.1.1 Major limitations

1. The existing hardware architecture has implementation only in VHDL
2. The existing hardware architecture cannot evaluate streams in a pipelined fashion in LLC

4.1.2 Minor limitations

4.1.2.1 Alternate evaluation of event-based and periodic streams

The *HLQInterface* in the HLC must carefully alternate between submitting pacings for event-based and periodic streams. This is necessary because these streams cannot be evaluated simultaneously, as the evaluation order in the existing implementation lacks the order between them. To resolve this, the *HLQInterface* enforces a specific ordering, event-based before periodic, at the hardware level. As a result, it needs to operate at twice the speed of other components in the HLC to ensure that both pacing signals are submitted to the queue in time. However, hardcoding this evaluation order into the architecture can limit flexibility and make it difficult to adapt to future changes in RTLola's evaluation semantics.

For example, let's consider the current semantics of evaluating event-based before periodic streams in RTLola as explained in Section 3.1.14.3. This is needed to break the race condition ocurring due to *hold* access as seen in Example a..1 in Section 3.1.14.3. However, *hold* access across event-based and periodic streams can also exist without the race-condition such as in example below:

Example 4.1.2.1.1 (Hold acess without race condition)

```
input x : Int
output a @1kHz := a.offset(by: -1).defaults(to: 0) + 1
output b := x + a.hold(or: 0)
```

Here, there is a clear evaluation order of $x \to a \to b$ as seen from the data-dependency. However, the current RTLola semantics doesn't respect this and does the evaluation in order $x \to b \to a$ as it always evaluates event-based before periodic. So, the input data we were to get at the exact time of periodic output such as at `0.001s`, would be missed here in the periodic output.

Arguably, we could enforce the ordering rule only when strictly needed such as to break the race-condition. The hardcoding of order in the existing architecture makes such changes difficult. From the design point, such ordering is better enforced in the evaluation order instead.

4.1.2.2 Timestamped inputs and determination of when to slide the sliding-window in LLC

The HLC annotates inputs with their corresponding timestamps to determine where each input falls within the sliding window span during evaluation. In the existing design, the decision to slide a sliding window is made locally within each sliding-window component in the LLC. These components keep track of the future time by which the sliding window must slide. They compare the current system time against the slide deadline to decide whether the window should advance.

This approach requires inputs to be timestamped and the system time to be tracked consistently across multiple components, namely the HLC, the queue, and the LLC.

However, one could argue that sliding window advancement is just another form of pacing. If so, it would be more coherent to delegate this responsibility to the HLC, alongside other pacing decisions. Doing so would simplify the overall architecture by eliminating the need to propagate timestamps across multiple components.

4.2 Objectives of the new architecture

The goal of the new architecture is to address the limitations identified in the existing design. While we retain the clear separation of concerns, where the HLC handles stream activation, and the LLC performs evaluation, we extend this architecture to overcome the previously discussed shortcomings and improve overall flexibility and efficiency. A key improvement in efficiency comes from introducing pipelined evaluation of streams in the LLC, enabling more parallelism and better resource utilization.

We focus on online monitoring, where hardware-based monitors provide the most value. Advanced RTLola features such as stream lifecycles and parameterization are beyond the scope of this architecture. Additionally, we concentrate on input and output streams, omitting trigger streams. In the context of hardware, trigger messages are less meaningful and can be effectively modeled as Bool-typed output streams instead.

4.3 Pipeline evaluation

Pipeline evalution is a way of processing data step by step like an assembly line. Each step does its part and hands the result to the next step. When the next step is working on the result, the first state can already start working on a newer data. This makes the system much more efficient as a step don't have to wait until all the following steps have finished their work.

Let us define pipeline evauation specifically in the context of our hardware monitor architecture.

Definition 4.3.1 (Pipeline Evaluation): Let X be the set of all possible input data in the system, $I = [i_0, i_1, i_2, ...]$ be an infinite input stream producing new input at each time step where $i_t \in X$, and $P = [P_1, P_2, ..., P_k]$ be the finite sequence of k computation stages.

The *pipeline state* $S(t)$ is a k-tuple representing the data currently in each stage: $S(t) = (s_1(t), s_2(t), ..., s_k(t))$ where $s_k(t)$ is the input data being processed by computation stage P_k at time t.

At time t, *input domain* $D_k(t)$ for stage P_k is the union of all potentially accessible data: $D_k(t) = H_k(t) \cup C_k(t) \cup I_k(t)$ where:
- Past pipeline states: $H_k(t) = \{S(i) \mid i \in \{0, 1, ..., (t - (k-1) - 1)\}\}$
- Output from preceeding stages: $C_k(t) = \{s_i(t - (i-1)) \mid i \in \{1, 2, ..., k-1\}\}$
- Input history: $I_k(t) = \left\{i_0, i_1, ..., i_{\text{t-(k-1)}}\right\}$

such that:
- $s_k(t) \subset D_k(t)$
- $P_k(s_k(t)) \subseteq s_{\text{k+1}}(t+1)$

The **pipeline evaluation** is the change of *pipeline state* over time such that:
1. Initial state, $S(0) = (s_1(0), \emptyset, \emptyset, ..., \emptyset)$
2. Fill state, When $t < k - 1$,
   $S(t) = (s_1(t), s_2(t-1), s_3(t-2), ..., s_t(1), s_{\text{t+1}}(0), ..., \emptyset)$
3. Steady state, When $t \geq k - 1, S(t) = \left(s_1(t), s_2(t-1), s_3(t-2), ..., s_{k(t-(k-1))}\right)$

In Definition 4.3.1, the system can receive a null input data into the system corresponding to absence of actual input. In such case, the computation stages might not have any input data thus producing a null output. Even so, the actual pipeline evaluation run continuously forever. The pipeline evaluation is defined such that each computation stage can take an input from the past evaluation results, delayed inputs, and the results from the earlier pipeline stages.

Definition 4.3.2 (Pipeline latency): The *pipeline latency* L is the time it takes for a single evaluation to flow through all the k pipeline stages:

$$L = k$$

Sometimes, after finishing working on a data, a pipeline step cannot immediately start working on a new data. It must wait for some time to let the results propagate. After waiting for this time, it is again ready to work on the new data.

Definition 4.3.3 (Pipeline wait): The *pipeline_wait* W is the amount of time (clock cycles) a pipeline step must wait after finishing working on its part of evaluation before it is ready to work again on a new data.

Please refer to Section 5.2 for better intuition behind *pipeline_wait*.

Definition 4.3.4 (Pipeline throughput): The *pipeline throughput* T is the number of evaluations completed per time step (clock cycles) once the pipeline has reached a steady state, with evaluations continuously flowing through all stages.

With *pipeline wait* W:

$$T = \frac{1}{1+W}$$

4.4 New hardware architecture

The new hardware architecture extends the existing RTLola hardware design [6]. Like the original, it is structured into three main components: the high-level controller (HLC), a first-in-first-out (FIFO) queue, and the low-level controller (LLC). The HLC is responsible for stream activation by generating pacing signals, while the LLC handles stream evaluation. This separation allows both components to operate independently and at different speeds.

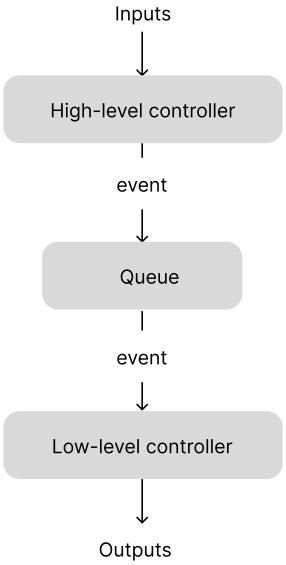


Figure 8: General architecture

The key innovation in the new architecture is the introduction of pipelined stream evaluation in the LLC. Unlike the existing design, where evaluations are performed sequentially, the new approach analyzes the stream dependency graph to enable safe and efficient pipelining of evaluations. The details of this dependency analysis are discussed in Section 5.

The existing architecture used the provided evaluation order from the RTLola frontend [73], which only provides the partial evaluation order (Section 5.1) of event-based and periodic separately. It lacks the ordering between event-based and periodic streams. The ordering between them is enforced by HLC by alternatively releasing pacing signals for event-based and periodic streams.

This alternate evalution doesn't go well when pipelining the evaluations in LLC. Pipelining benefits significantly from a single, unified evaluation order in which streams are processed sequentially, one after another. For this reason, we do not use the evaluation order provided by the RTLola frontend.

RTLola frontend [73] provides an intermediate representation called RTLolaMIR, which captures all relevant information from the RTLola specification, including stream pacing, data dependencies, and data types. Based on this information, we perform our own graph analysis to derive a total evaluation order (Section 5) that includes both event-based and periodic streams in a unified sequence. This eliminates the need for alternate evaluation altogether.

4.4.0.1 High-level controller (HLC)

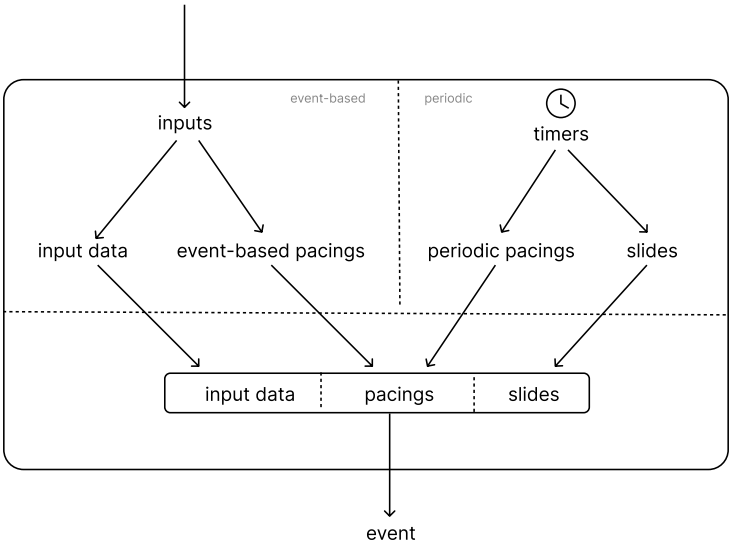


Figure 9: High-level controller (HLC)

The High-Level Controller (HLC) running in a pipeline fashion is responsible for capturing system inputs and activating streams by issuing the appropriate pacing signals. For event-based streams, pacing signals are generated based on logical conditions over input presence. For periodic streams, the HLC must track time to activate streams at the correct intervals.

Unlike the existing architecture, our design also handles sliding-window advancement within the HLC. By treating the timing to slide a window as just another type of pacing, the HLC centralizes all time-related logic. As a result, there is no need to timestamp input data, and time no longer needs to be propagated across the system.

Internally, the HLC contains dedicated timer circuits for each unique period. If multiple periodic streams or sliding windows share the same period, they use a common timer. These timers accumulate time by incrementing each clock cycle. Once a timer exceeds its predefined threshold, it triggers a pacing signal and resets.

The HLC generates three main information:
1. The actual input data
2. Pacings for event-based and periodic streams
3. Pacing for when to slide the sliding windows

Unlike in the existing architecture, all of this information is submitted together as a single unified event. Whenever any of these actions occur, an event — an encapsulated structure containing both data and control information — is generated and passed to the queue. Each event represents everything that happened during a given cycle and contains all the information necessary for correct evaluation.

By handling all timing and pacing logic in the HLC, downstream components like the queue and LLC become agnostic to whether a stream is event-based or periodic. There's no need to timestamp inputs or propagate time through the system. Combined with a unified total evaluation order, this design removes the need for alternating evaluation and avoids the tight coordination requirements between components found in the existing architecture.

Each part of the system can now operate more independently, without needing precise timing alignment to ensure correctness.

4.4.0.2 Queue

Both the high-level controller and low-level controller run in a pipelined fashion. However, they are not synchronized. The low-level controller goes through a step-by-step evaluation process, and depending on data propagation time, it might need to wait (*pipeline_wait*) until the data is evaluated at the source, causing a delay before the next evaluation cycle can run. This contrasts with the high-level controller, where periodic timeouts or input data can arrive at any time. Hence, it is not possible to synchronize them.

For example, consider a case where the monitor is receiving inputs at such a high rate that, in each cycle, we have a new input. In this case, the High-Level Controller must produce a valid event every cycle to report the new data and pacing. If we don't do this, we will lose input data.

Hence, we need a buffer between them. Furthermore, since the events in the buffer must be processed in the order they arrive, the buffer must be a first-in-first-out queue. The queue helps in ordering events and provides a buffer to absorb rapid bursts of events, provided it has enough memory.

During an overflow, the queue rejects new events while keeping existing events intact. This ensures that the monitor continues to produce accurate results for already confirmed events, as no historical data is lost. Once queue space becomes available, the system can resume normal operation.

Remark 4.4.0.2.1 (Queue protects existing evaluations by rejecting push when full). *During an overflow, the queue rejects new events to keep existing ones intact. This ensures that the monitor continues to produce accurate results for already confirmed events, as no historical data is lost.*

4.4.0.3 Low-level controller (LLC)

The low-level controller pops the event from the queue and evaluates the streams accordingly in the correct order in a pipeline fashion. It is constructed based on the result from the dependency graph analysis (Section 5).

The graph analysis yields three key pieces of information:
1. Total evaluation order including both event-based and periodic streams (Section 5.1)
2. The necessary `pipeline_wait` (Section 5.2)
3. The amount of result history each evaluation node must maintain (Section 5.3)

The evaluation order defines the sequence of pipeline steps. Streams that appear at the same position in the evaluation order are evaluated together in the same pipeline stage. The `pipeline_wait` specifies how many cycles a pipeline stage must wait after producing a result before it can start processing new data. The third piece of information, the result history, indicates how much memory each stream needs to retain so that its successors can

still access the necessary values even after the stream has moved on to processing new inputs.

The required result history of stream is bounded by the data dependency and the *pipeline_wait*. So, the required memory for it is finite.

The final step of the pipeline is the output stage, where all results from the individual pipeline steps are collected and emitted from the system. It is done this way to synchronize the outputs.

Remark 4.4.0.3.1 (Evaluation order includes sliding windows). *The evaluation of periodic streams with sliding windows involves two steps. First, the sliding window buckets must be updated with new data. Only after that can the periodic stream be computed by aggregating the buckets. This means that, in hardware, evaluating such streams would typically require two cycles.*

To simplify this, we include the sliding windows explicitly in the evaluation order. As a result, each item in the evaluation order — whether a stream or a sliding window update — takes exactly one cycle to evaluate. This approach simplifies hardware implementation, especially when using pipelining.

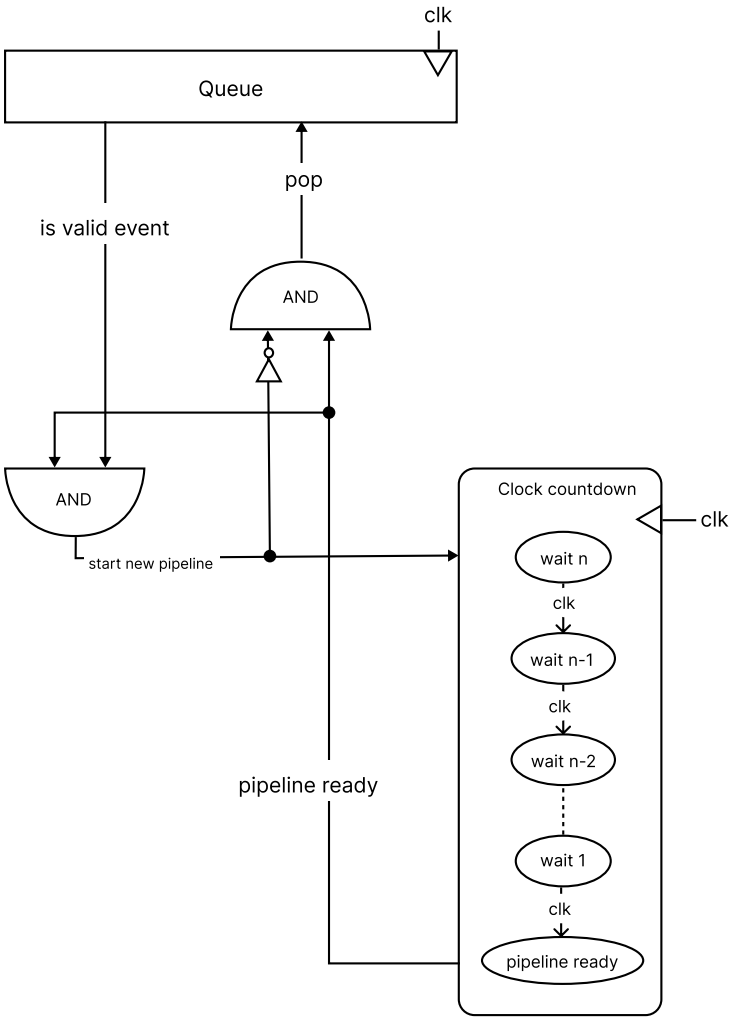


Figure 10: Event pop mechanism in low-level controller

Figure 10 shows the concept circuit of how the low-level controller pops events. When ready, the low-level controller pops an event from the queue. If there is a valid event in the queue, it starts the evaluation pipeline; otherwise, it keeps trying to pop events from the queue. After starting the pipeline evaluation cycle, it waits according to the `pipeline_wait`, which could be zero. After waiting the required number of cycles, the LLC will be ready to pop and begin the next evaluation cycle.

For the evaluation of streams, the popped event contains all the necessary information, such as pacing signals indicating which streams need to be evaluated, slide signals showing which sliding windows need to advance, and the actual input data.

For example, consider an evaluation order of $x \to a \to b \to c$ with a `pipeline_wait` of 2. Depending on the events from the queue, the evaluation could proceed as follows:

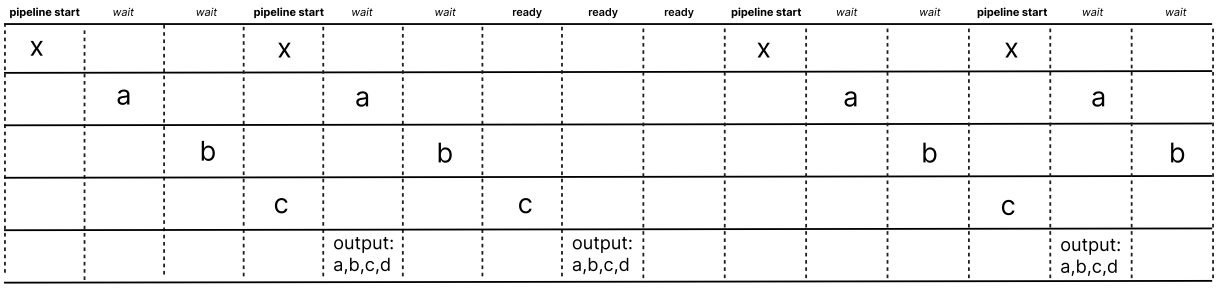


Figure 11: Example evalution in LLC

Here considering the first pipeline, pacings have enabled all the streams `x`, `a`, `b`, `c`. The evaluation happens in the order of $x \to a \to b \to c$. After evaluating all streams, the outputs are produced in the following cycle. During the evaluation of the first cycle, the second evaluation cycle was started in parallel after waiting two cycles as necessary. However, after the second pipeline cycle, there was no valid event for some time, so the third evaluation cycle started much later.

a. Tagging the results for easy access of offset values

For the easy access of past offset values, we tag the results in the streams. Each stream has its own tag that starts from one. On each successive evalution the result is tagged with the tag one value higher. With it, any past offset value can be easily matched with the past tag. The value of tag keeps increasing this way until a maximum tag is reached. After which it agains starts back from one. The maximum value of tag is chosen higher than any offset or the span of the sliding window to ensure that there are no duplicate values with the same tag in the window. This design decision greatly simplifies matching the right value.

For example, consider `output a := c.offset(by: -3).defaults(to: 0)`.

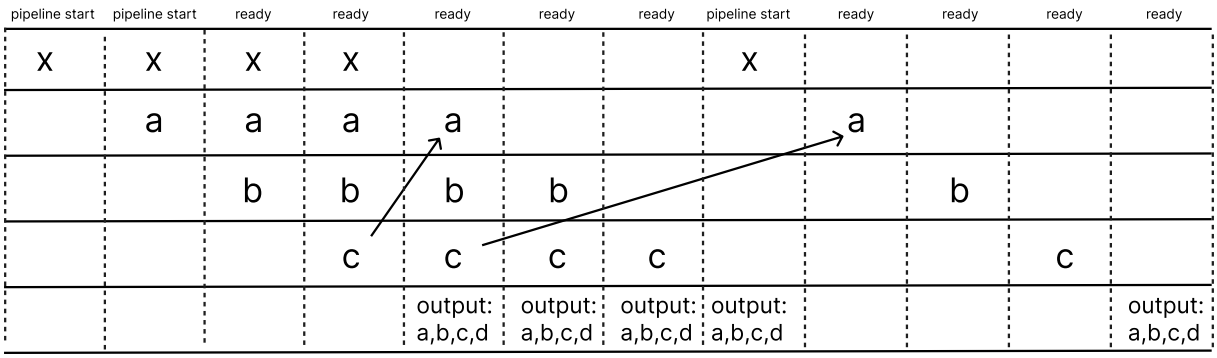


Figure 12: Example showing use of tags in LLC

In Figure 12, the node `a` depends on the value of `c` from three evaluation cycles ago (offset 3). In the fourth evaluation cycle, the arrow pointing to `a` represents this offset access - meaning `a` is using the value of `c` from three cycles prior. In this case, we only need to remember the latest value of `c`, since that value happens to be at the required offset.

However, consider the fifth evaluation cycle, which occurs after a few idle cycles (i.e., without any new events). Now, to access the value of `c` from three cycles ago, we must retain at least the last three values of `c`.

We see that the position of the required value in the storage window changes between the two evalution cycles:

- In the fourth cycle, the required value is at the most recent position in the window
- In the fifth cycle, it's two positions earlier in the window

This illustrates the advantage of tag-based matching: the exact position of the value within the window doesn't matter. As long as the tag matches, the correct value of `c` can be retrieved — regardless of where it sits in memory.

4.5 Limits of the new architecture

4.5.1 Input frequency

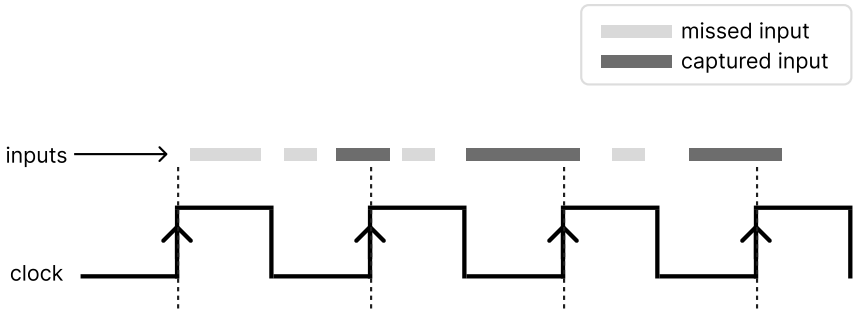


Figure 13: Input frequency limited by clock frequency

To obtain a hardware in the real-world, we need to know the memory boundary of hardware and the frequency for the clock. The chosen clock frequency determines the input frequency we can support. As the inputs are read synchronized by the clock, the input signal must be sustained until the edge of the clock. Any input that comes in between will be missed as seen in the example in Figure 13.

4.5.2 Memory of queue

Another important factor limiting the input frequency is the discrepency in the speed of HLC and LLC. When the next evaluation cycle in LLC can only be started after waiting some time (`pipeline_wait`), the speed of LLC is `1 + pipeline_wait` times slower than HLC. If the HLC always works on its maximum capacity then LLC will never be able to catch up requiring infinite memory in the queue to buffer the pending events. Therefore, the size of queue is determined by the `pipeline_wait` and the maximum burst of inputs the monitor must support.

Assuming an empty queue, in order to support a maximum burst of `n` inputs coming at each clock - we need the queue size of at least:

$$\text{min\_queue\_size} = \begin{cases} 1 \text{ if pipeline\_wait} = 0 \\ n - \left\lfloor \frac{n}{1+ \text{ pipeline\_wait}} \right\rfloor \text{ else} \end{cases}$$

When the `n` burst of inputs have arrived, the LLC already manages to finish $\left\lfloor \frac{n}{1+ \text{ pipeline\_wait}} \right\rfloor$ events, hence the substraction in the 'else' case. Whereas in the case when the `pipeline_wait = 0`, the HLC and LLC run equally fast so we can completely get rid of

queue to interface them. However, for simplity we keep the queue in place to interface them so it needs at least a single unit memory for event to interface them.

4.5.3 Output latency

The monitor cannot produce an output immediately upon receiving an event - there is some inherent latency. This delay is due to the step-by-step evaluation of streams in the LLC, following a defined evaluation order. Additionally, the `pipeline_wait` affects latency by determining when an event from the HLC can begin processing in the LLC. Overall, the latency is bounded by the length of the evaluation order and the value of `pipeline_wait`, ensuring that each event produces an output within a finite number of clock cycles.

4.5.4 Periodic frequency

The maximum periodic frequency the monitor can support is limited by the output latency. If the periodic frequency exceeds the output latency, the HLC may submit a new periodic pacing before the LLC has finished processing the previous one. As a result, events will arrive too quickly for the LLC to handle, eventually causing the queue to overflow.

Chapter 5

Dependency Graph Analysis For Pipeline Evaluation

5.1 Evaluation order

Evaluation order has been defined in background Section 3.1.13 based on past works. We want to redefine it in the context of our prototype for better clearity.

Note that the RTLola streams must be evaluated in the order corresponding to the dependency between them given by the dependency graph (Section 3.1.11). Following the chain of strict dependency i.e without past offsets (Section 3.1.6), a partial order can be built. These partial orders can be indepedently evaluated so any interleaving between them is fine as long as the total order obtained from interleaving respects the individual partial orders.

Definition 5.1.1 (Partial evaluation order): On a *dependency graph* $G = \langle V, E \rangle$, the *partial evaluation order* P is a sequence of disjoint subsets of vertices:

$$P = [S_1, S_2, S_3, ..., S_t]$$

such that:
1. for $i \neq j, S_i \cap S_j = \emptyset$,
2. $\forall e = \langle u, v, w \rangle \in E$ with $w = 0$, $u \neq v$ if $u \in S_i$ and $v \in S_j$, then $u \rightarrow v$

In Definition 5.1.1, each set of vertices in the evaluation order represents nodes that can be evaluated simultaneously. The second rule ensures that for accesses without offsets, the producer node must be evaluated before the consumer node. When a negative offset is used, the consumer relies on a past value from the producer, which is already available. In such cases, a strict evaluation order is unnecessary — any order between the two nodes is valid. Although future offsets are not currently supported in RTLola, they can be reasoned about in an analogous way.

Definition 5.1.2 (Total evaluation order): On a *dependency graph* $G = \langle V, E \rangle$, the *total evaluation order* O is the interleaving of all disjoint *partial evaluation orders* giving a sequence of disjoint subsets of vertices:

$$O = [L_1, L_2, L_3, ..., L_k]$$

such that:

1. $\bigcup_{i=1}^{k} L_i = V$,
2. for $i \neq j, L_i \cap L_j = \emptyset$,
3. With a *partial evaluation order* P,
   if $v_i \in S_a, v_j \in S_b, S_a \in P, S_b \in P$ and $a < b$, then
   $\exists L_p \in O, \exists L_q \in O, v_i \in L_p, v_j \in L_q$ such that $p < q$.

The third rule in Definition 5.1.2 ensures that the total order respects the order of all partial orders. There can be many valid total order as long as they respect the rules.

For example: The total order of $[\{a, d\}, \{b\}, \{c, e\}]$ can be obtained by interleaving the partial orders $[\{a\}, \{b\}, \{c\}]$ and $[\{d\}, \{e\}]$. However, many other total orders are equally valid — such as $[\{a\}, \{b\}, \{c, d\}, \{e\}]$, $[\{d\}, \{a\}, \{e\}, \{b\}, \{c\}]$, etc.

5.2 Simple idea behind pipeline evaluation

Let's get an intuition on how the pipeline evaluation could work with an actual example.

Example 5.2.1 (Idea behind pipeline evaluation)

```
input x: Int
output a := x + 1
output b := a + d.offset(by: -3).defaults(to: 0)
output c := b + 1
output d := c + 1
```

The data-flow of the spec, can be understood by the graph:

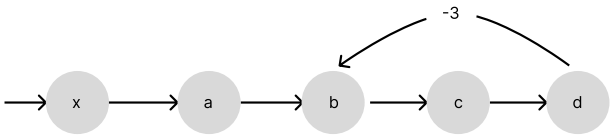


Figure 14: data-flow graph

From the data-flow graph we can see that the evalution must follow the order:
$x \to a \to b \to c \to d$

Remark 5.2.1 (Data-flow graph). *A* data-flow graph *is a representation of the* dependency graph *showing how data values move and how they depend on each other.*

The evaluation order can be pipelined like:

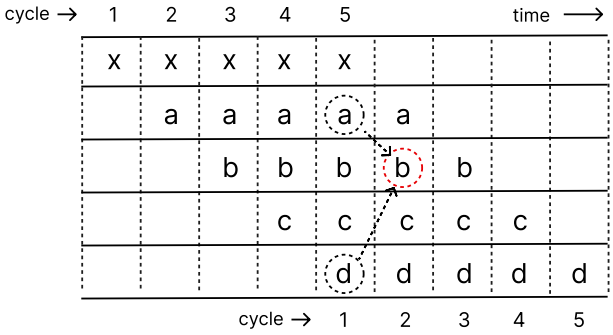


Figure 15: pipeline evaluation

In Figure 15, columns represent a time-unit i.e one clock cycle. Rows represent a pipeline step. The stream nodes are ordered in the pipeline steps as per the evaluation order.

We can observe that when the stream/node `b` is evaluated, all the values it depends on — namely, the latest value of `a` and a past value of `d` — are already available. Therefore, the evaluation order can be pipelined as illustrated in the figure.

However, if `b` were to depend on a more recent value of `d`, pipelining would no longer be immediately possible. Consider the scenario shown in Figure 15: if `b` depended on `d` with only a one offset into the past instead of three, then during the evaluation of `b` at evaluation cycle *4* (as circled in the figure), the required value of `d` i.e corresponding to the eval cycle *3* would not be available until the next two cycles. As a result, the pipeline must wait.

> **Remark 5.2.2** (Evaluation cycle). *By evaluation cycle, we mean the complete evaluation of all nodes corresponding to the input. The evaluation cycle finishes only when all the steps have finished working on the results.*

That is, with the output `b` as:

```
output b := a + d.offset(by: -1).defaults(to: 0)
```

The evaluation order can be pipelined best as:

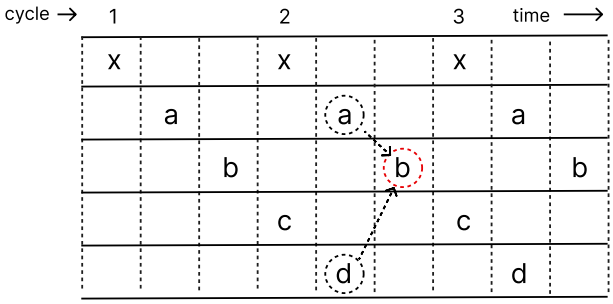


Figure 16: pipeline evaluation with offset (-1)

With Figure 16, when `b` is to be evaluated the the required past value of `d` is already available. Hence, the evaluation for `b` can proceed forward.

By waiting two cycles before starting the next pipeline cycle i.e *pipeline wait* of 2, the system gave enough time for the propagation of data.

Following this observation, we see that the evaluation can be done in the data-flow order of the graph, and depending on the -ve offset we have to wait until we can run the next pipeline. In the worst case, we have to wait until all the nodes are evaluated which would result in the evaluation without pipelining just like in the existing *RTLola to VHDL compiler* [6]. In the best case we get pipeline_wait $= 0$, giving us the maximum throughput (Definition 4.3.4).

5.3 Memory implications of pipelining

During pipeline evaluation, multiple evaluation cycles can be in progress at the same time, each in a different stage. As the node start operating for the next evaluation cycle, the old values must be remembered until they are consumed which could be later in the future. This creates a need to remember results from various evaluation cycles.

For example consider the following pipeline evaluation corresponding to the RTLola specification:

```
input x : Int
output a := x + 1
output b := a + 1
output c := x + a + b
```

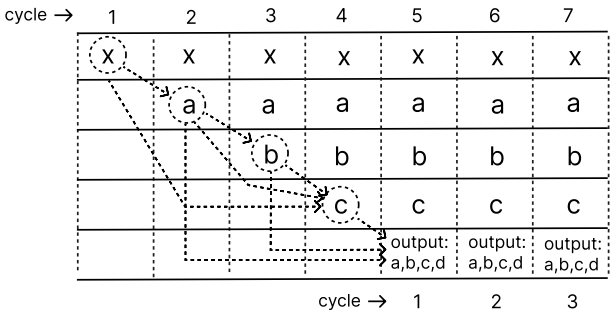


Figure 17: Memory implications of pipelining

Here, in Figure 17, we evaluate all the nodes in their evaluation order and then output the result when all the output nodes have been evaluated. The last `output` step is the final step when the results of individual evalutions is outputed corresponding to the evaluation cycle.

In the figure, the dotted lines indicte the data-flow. From which, for example, we see that the value of `x` is consumed by `c`. But, by the time `c` corresponding to the first evaluation cycle consumes `x` i.e the circled `c` in cycle `4`, the `x` has already evaluated values correponding to newer evaluation cycles `2` and `3`. This means that at least the last three values of `x` must be remembered. Following the same line of reasoning, in order to be able to output the results in the final output step, we need to remember the latest three values of `a`, latest two values of `b`, and only the last one value of `c`.

Remark 5.3.1 (Tight pipeline evaluation). *The pipeline evaluation shown in Figure 17 demonstrates tightly packed evaluation cycles i.e each pipeline stage processes a new evaluation in every clock cycle. This represents a case where data arrives frequently, allowing continuous processing without having to wait for input. However, in practice, there may be periods when the system receives no input, resulting in idle cycles without any evaluation.*

5.4 Obtaining evaluation order

Let's figure out evaluation order from the dependency graph looking into individual subproblems before generalising the algorithm.

5.4.1 Evaluation order from a directed acyclic graph (DAG)

On a directed acyclic graph (DAG), we can perform level-order traversal to get a set of nodes in the same level and filter them by reachability. This way we can obtain the evaluation order from the graph.

Abscence of cycle in a DAG means there is no dependency on the result of the future node so the next pipeline can immediatley start without waiting.

Remark 5.4.1.1 (Directed acyclic graph (DAG)). *A directed graph with no cycles is called a directed acyclic graph (DAG).*

Remark 5.4.1.2 (Level order traversal). *Level order traversal, also known as breadth-first search is a method of visiting all nodes by starting at the root and exploring all nodes at each depth before moving to the next.*

Example 5.4.1.1 (DAG)

```
input x : Int
output a := x + 1
output b := a + 1
output c := x + a + b
```

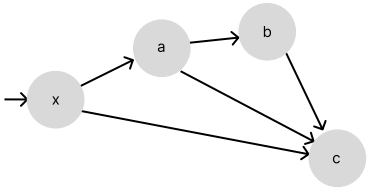


Figure 18: DAG

In Example 5.4.1.1, starting at roots `[x]` the next level of nodes is `[a, c]`. However, the node `c` is reachable from `a` along the path $a \to b \to c$. As `c` must come later we remove it from this level giving us `[a]`. We collect all the children from the nodes in the current level to obtain the next level. The next level from `[a]` would be `[b, c]`. As `c` is rechable from `b` we remove `c` from this level as well giving us `[b]`. Finally the last level is `[c]`. This gives us the order of levels: $[x] \to [a] \to [b] \to [c]$ which is the evaluation order we must follow.

Algorithm 5.4.1.1 (Evaluation order from DAG)

```
DagEvaluationOrder(roots):
 1  order ← empty list of lists
 2  current_level ← roots
 3  next_level ← ∅
 4  while current_level ≠ ∅ do
 5      append current_level to order
 6      for node in current_level do
 7          for child in GetChildren(node) do
 8              next_level.add(child)
 9      current_level ← RemoveReachableRoots(next_level)
10      next_level ← ∅
11  return order
```

```
RemoveRechableRoots(roots):
1  all_reachables ← ∅
2  for node in roots do
3      reachable ← ReachableNodes({node})
4      reachable ← reachable \ {node}                  // exclude the node iteself
5      all_reachables ← all_reachables ∪ reachable     // union
6  return roots \ all_reachables                       // exclude all rechable nodes
```

```
ReachableNodes(roots):
1  reachable ← ∅
2  stack ← roots
3  while stack is not empty do
4      node ← stack.pop()
5      add node to reachable
6      for child in GetChildren(node) do
7          if child ∉ reachable then
8              stack.push(child)
9  return reachable
```

5.4.2 Evaluation order from a cyclic graph

In Section 5.4.1, we discussed how to determine the evaluation order when the dependency graph is acyclic. However, dependency graphs derived from RTLola specifications may contain cycles in various forms. These different types of cycles are explored in detail in the background chapter in Section 3.1.14. The cases are:

1. Cycles in event-based streams via negative offsets
2. Cycles in periodic streams via negative offsets
3. Cycles across event-based and periodic streams via holds

5.4.2.1 Removing cycles due to offsets

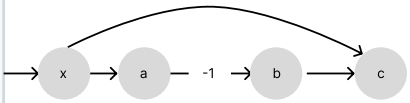
If cycles within event-based and periodic nodes can exist only via -ve offsets, an obvious way to obtain the evaluation order would be to simply ignore them. This way the graph can be converted to DAG and we can use the algorithm for DAG to obtain the evaluation order.

Negative offsets represents values from the past. During the current evaluation cycle, we can assume that these past values are already available. While this assumption may not strictly hold in a pipelined evaluation, it becomes valid if a sufficient `pipeline_wait` is introduced. As a result, negative offsets do not impose constraints on the evaluation order and may be ignored when determining it.

However, we must be careful not to ignore all negative edges when calculating the evaluation order, as negative offsets can exist in the graph without necessarily forming a cycle.

Example 5.4.2.1.1 (-ve offset without creating a cycle)

```
input x : Int
output a := x + 1
output b := a.offset(by: −1).defaults(to: 0)
output c := x + b
```



Therefore, we must be careful to ignore only those negative offset edges that actually contribute to a cycle in the graph.

5.4.2.2 Removing cycles due to holds

The third kind of cycle in RTLola across event-based and periodic streams occur via `hold` access (Section 3.1.14.3). As the hold acees can lead to a cycle with race-condition, RTLola follows a rule of evaluating event-based streams before periodic streams (see Section 3.1.14.3).

To deal with such cycles we can split the graph into two subgraphs, one containing only event-based nodes and another containing only periodic nodes. The *partial evaluation orders* (Definition 5.1.1) from each groups can later be combined folllowing the *event-based before periodic* rule to obtain the *total evaluation order* (Definition 5.1.2).

5.4.2.3 Idea of obtaining the evaluation order from a cyclic graph

Based on these observations, we can follow the idea below to determine the evaluation order.

1. Break cycles across event-based & periodic streams by splitting graph into only event-based & periodic streams
2. Within each split group, break cycle due to -ve offset by ignoring -ve offsets
3. Obtain the partial evaluation order from the resulting DAGs
4. Merge the partial evaluation orders correctly to get the total evaluation order

5.5 Optimization of partial evaluation order with offsets

One interesting observation regarding -ve offset nodes is that, they can potentially be evaluated much earlier in the evaluation order. For better illustration, let's consider an example:

```
input x : Int
output a := x + 1
output b := a + 1
output c := b.offset(by: -2).defaults(to: 0) + 1
output d := c + 1
```

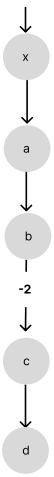


Figure 21: Negative offset in a DAG

In Figure 21, `c` accesses a past value of `b`. As explained in Section 5.4.2.1, this past result will already be available during the current evaluation cycle, provided we introduce sufficient `pipeline_wait`. This implies that the $c \rightarrow d$ evaluation can occur earlier, as it doesn't depend on the current data flow from $x \rightarrow a \rightarrow b$.

However, the amount of `pipeline_wait` required varies depending on when we evaluate `c`. Let's compare different ways we could structure the $c \rightarrow d$ evaluation order as follows:

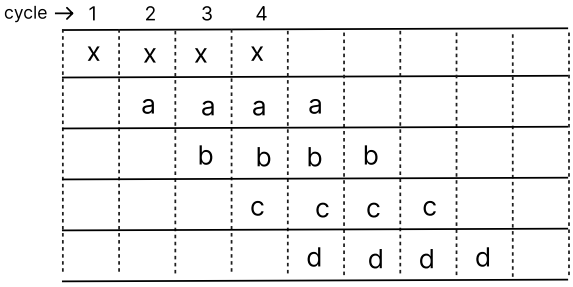


Figure 24: without optimization

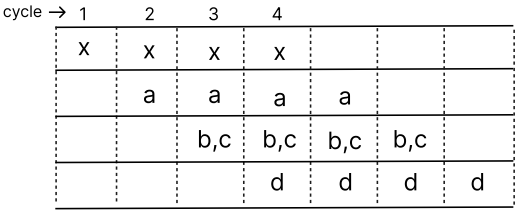


Figure 25: pulling $c \rightarrow d$ by 1 step

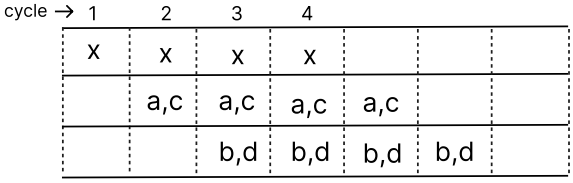


Figure 26: pulling $c \rightarrow d$ by 2 steps

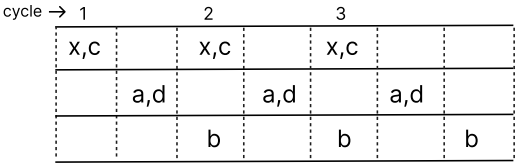


Figure 27: pulling $c \rightarrow d$ by 3 steps

The figure in Figure 24 shows the evaluation without any changes. In the subsequent figures, the $c \rightarrow d$ is pulled earlier by various steps to explore potential optimizations.

Comparing Figure 25 with Figure 24, we see a reduction in the total number of pipeline levels, which is already an improvement, as the result can be produced sooner. Fewer levels are also beneficial for memory usage, since we need to store fewer intermediate results until the output phase at the end of the pipeline.

If we evaluate `c` one step earlier, as in Figure 26, the number of levels decreases even further. However, pulling `c` even earlier — as in Figure 27 — requires a `pipeline_wait` of one cycle. This is because we now need to wait for the value of `b` to become available before evaluating `c`.

Among these options, Figure 26 is clearly the best. It achieves the tightest pipelining with zero `pipeline_wait`, while also minimizing the number of pipeline levels, implying less memory requirement for storing intermediate results until the output phase.

5.6 Trade-offs and pitfalls in evaluation order optimization

Choosing the best evaluation order is a balancing act between pipeline and memory trade-offs. Optimizing for memory can hamper pipelining, and vice versa. We might optimize the pipeline for one locality, which in turn can conflict with local optimization at another point — leading to a worse global outcome. The same applies when optimizing for memory.

To better understand some of these challenges, let's consider few scenarios here:

5.6.1 Offsets tug of war

There are cases where optimizing for one dependency conflicts with the optimization for another. This apparant tug of war between different poles of attraction can be better understood by the example below:

```
input x : Int
output a := x
    + c.offset(by: -1).defaults(to: 0)
output b := a + 1
output c := b
    + f.offset(by: -1).defaults(to: 1)
output d := c + j
output e := d + 1
output f := e + 1
output g := b + 1
output h := g + 1
output i := h + 1
output j := i + 1
```

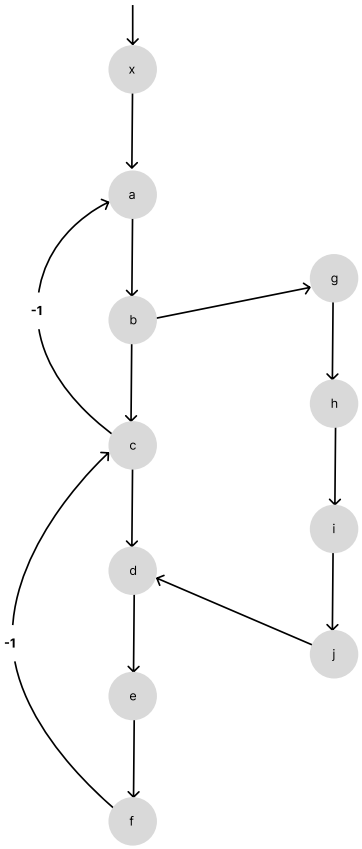
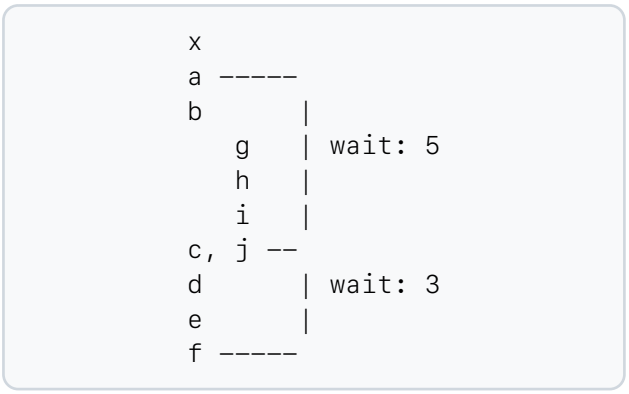
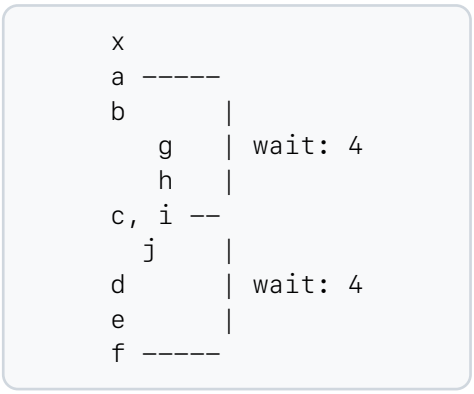


Figure 28: Offsets tug of war

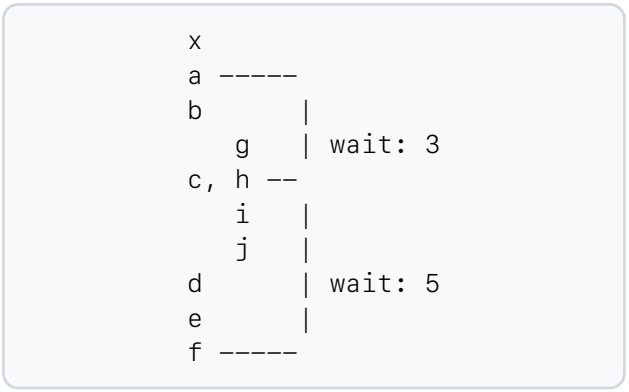
In Figure 28, we have two parallel dataflow that happens in between `b` and `d`. As there is no dependency between them, they could be ordered in various ways.



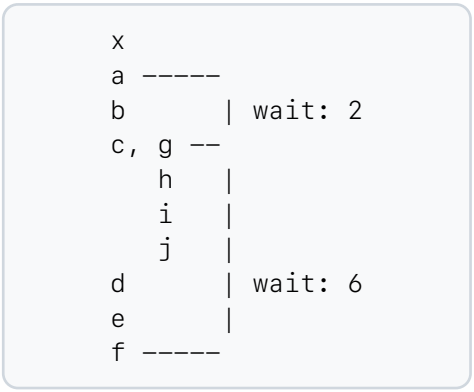
Evalution order option: 1



Evalution order option: 2



Evalution order option: 3



Evalution order option: 4

Figure 30: Different evaluation order options

5.6.1.1 Implication for `pipeline_wait`

Figure 30 shows various options of evaluation order for Figure 28. In Option 1, we evaluate `c` together with `j`. This places `c` as close as possible to `f`, which depends on it. As a result, the data propagation time from `c` to `f` is minimized, requiring only 3 cycles i.e suggesting `pipeline_wait` of 3. However, `a` also depends on the value of `c`. So, by pulling `c` closer to `f`, we move it further away from `a`, increasing the data propagation time in that path suggesting `pipeline_wait` of 5.

This tug-of-war between dependencies makes it challenging to choose the right evaluation order. Ultimately, it is the maximum propagation time that matters for overall `pipeline_wait`. Therefore, Option 2 from Figure 30 is the best choice in terms of pipelining.

5.6.1.2 Implication for memory

It is important to note that the distance between nodes with dependencies directly influences memory. Since the source node needs to remember the result until it is consumed, increasing the distance also increases the time it must keep that result.

Each evaluation node might work with different sizes of data — for example, one might be a periodic stream processing a large number of sliding window buckets, while another might handle just a single boolean.

Therefore, it might be the case that options like Option 4 from Figure 30 are more desirable in terms of memory if the result in `c` has a large memory footprint, even though it is worse than Option 2 in terms of `pipeline_wait`.

5.6.2 Stretching of an order

Between parallel evaluation orders in branches, we could arrange them in various ways to choose the best order interms of our pipeline and memory priority. When the length of these parallel branches is different, it creates an interesting situation where stretching the shorter branch can lead to a better overall evaluation order.

For example:

```
input x : Int
output a :=
  c.offset(by: -1).defaults(to: 0) + x
output b := a + 1
output c := b + 1
output d := b + 1
output e := c + 1
output f := d + 1
output g :=
  l.offset(by: -1).defaults(to: 0) + e
output h := f + 1
output i := g + k
output j := h + 1
output k := j + 1
output l := i + 1
```

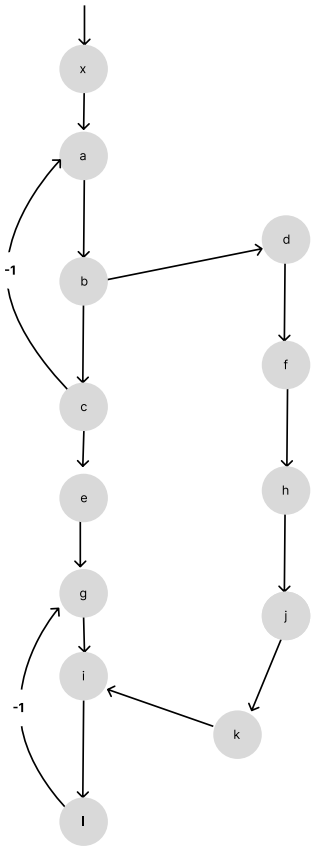


Figure 31: Stretching of an order

In this example, in order to reduce the propagation time, we can pull the `c` as close as possible to `a` and pull `g` as close as possible to `l`, effectively stretching the order $c \to e \to g$. For `e`, we have different choices of putting it in different levels between `c` and `g`. Let's consider one of the evaluation order as below:

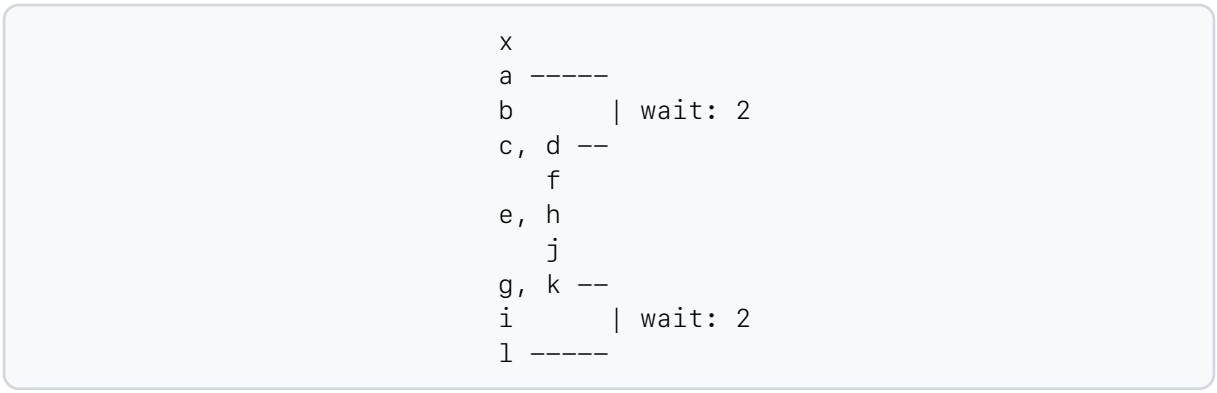


Figure 33: Stretching of order $c \to e \to g$

5.6.2.1 Implication for `pipeline_wait`

The evaluation order in Figure 33 gives us the minimum `pipeline_wait`. The position of `e` between `c` and `g` does not affect the `pipeline_wait`, as there is no cycle involving those nodes.

5.6.2.2 Implication for memory

While the evaluation order in Figure 33 is optimized for minimal `pipeline_wait`, it may not be optimal in terms of memory. For example, since the result of `c` is consumed by `e`, it must be stored until `e` is evaluated. Similarly, the result of `e` must be remembered until `g` is evaluated. This creates a tug-of-war situation for `e` in terms of memory usage. The actual impact depends on the data size of the results produced by `c` and `e`.

5.7 Heuristic algorithm to find evaluation order

The evaluation order algorithm we use is based on the observations discussed earlier. We first split the graph into multiple components that can be evaluated as DAGs. Then, we merge the evaluation orders from these components to produce a single final evaluation order.

As described in Section 5.4.2.3, we break cycles by first dividing the graph into two groups: event-based nodes and periodic nodes. We then further split the graph by ignoring the negative offsets resulting to cycles. The evaluation order from a DAG is obtained following the idea as shown in Example 5.4.1.1.

For simplicity, we do not allow *stretching of orders* within branches, as discussed in Section 5.6.2. This choice limits some optimization opportunities, but keeps the algorithm simpler.

5.7.1 Merging evaluation orders

The individual evaluation orders are merged to produce a total evaluation order. First, we merge the orders within the event-based group and within the periodic group separately. Then, we combine the resulting event-based and periodic orders into a single final evaluation order.

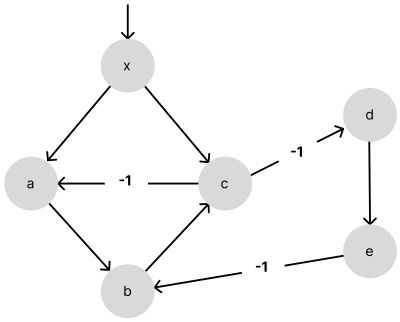
5.7.1.1 Merging within event-based and periodic groups

In order to merge orders within the same group, we need to take the negative offsets into account. As shown in Section 5.5, the way we merge these orders can lead to significantly different `pipeline_wait` and memory requirements.

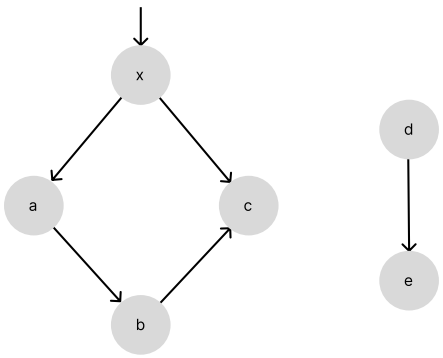
For simplicity, we do not allow stretching of orders, as discussed in Section 5.6.2 — instead, we keep the orders tight. However, we still aim for some optimization by exploring different ways the individual orders can be merged. For each combination, we calculate the resulting `pipeline_wait` and memory usage, and choose the one with the lowest `pipeline_wait`. If multiple combinations result in the same `pipeline_wait`, we break the tie based on total memory usage — favoring lower memory. In other words, we prioritize pipeline throughput over memory efficiency.

Example 5.7.1.1.1 (Example specification showing merge algorithm)

```
input x : Int
output a :=
  c.offset(by: -1).defaults(to: 0) + x
output b :=
  e.offset(by: -1).defaults(to: 0) + a
output c := x + b
output d := c.offset(by: -1).defaults(to: 1)
output e := d * 2
```



Let's consider Example 5.7.1.1.1 to illustrate it better. Here, we get the following subgraphs and individual eval orders after ignoring the -ve offsets.




```
Eval order 1:
x -> a -> b -> c

Eval order 2:
d -> e
```

We merge the individual eval orders by checking each possible combination without stretching. In the example, we can merge in the following ways:

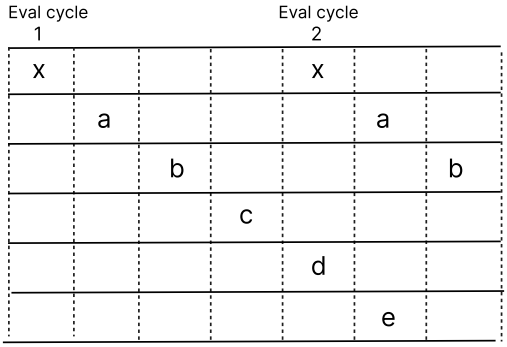


Figure 36: Merge option 1

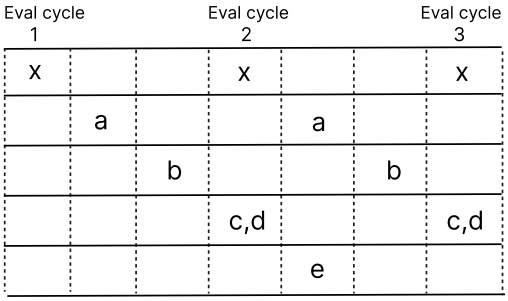


Figure 37: Merge option 2

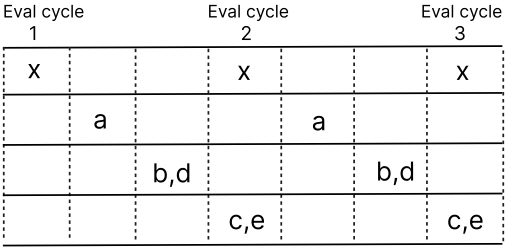


Figure 38: Merge option 3

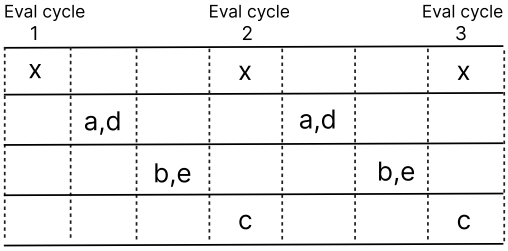


Figure 39: Merge option 4

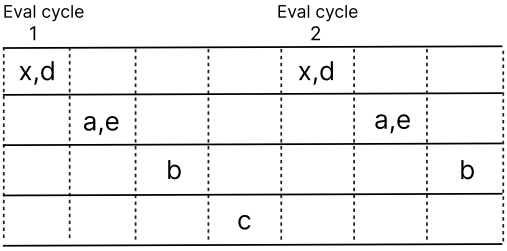


Figure 40: Merge option 5

Out of all options, the options 3 and 4 are most desirable as they have the lowest `pipeline_wait` as well as the lowest levels in the evaluation order. We break the tie between them based on the total memory, however they are also equally good here. As a final tie breaker, we choose to evaluate nodes as early as possible. Therefore, option 4 would be our evaluation order of choice.

5.7.1.2 Merging the event-based and periodic orders together

To merge the evaluation orders of event-based and periodic streams, we need to consider `hold` accesses across the two groups. As discussed in Section 3.1.14.3, RTLola enforces a rule that event-based streams must be evaluated before periodic streams to avoid race conditions caused by `hold`. This ordering must be preserved when merging the two groups.

A straightforward and valid merge is to simply append the periodic evaluation order after the event-based order. This is essentially what the existing compiler does by alternately evaluating event-based and periodic streams.

However, not all nodes between the two groups are connected by edges. So, we can potentially evaluate periodic order earlier. To determine how early this can be done, we can

iteratively try shifting the periodic order earlier, one level at a time, until we reach a point where the event-based before periodic rule is violated. To detect violations, we examine all edges that cross between the two groups and ensure that the event-based node associated with each edge is evaluated before the corresponding periodic node.

Remark 5.7.1.2.1 (Binary search for merging). *Since the search space for merging — while respecting the event-based before periodic rule — is monotonic, we use binary search instead of a linear search to improve efficiency.*

Example 5.7.1.2.1 (Example showing merge between groups)

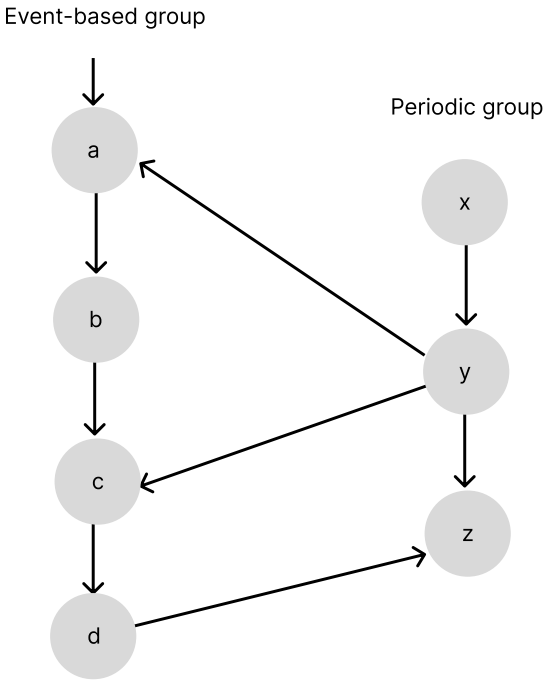


Figure 41: Graph

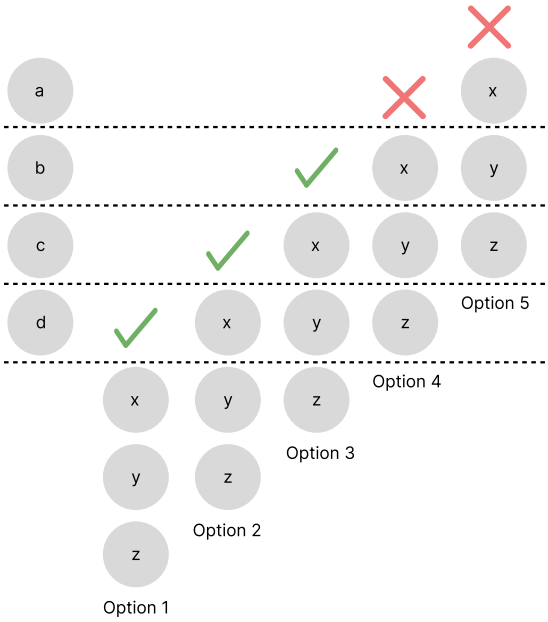


Figure 42: Merge options

For example, consider Example 5.7.1.2.1, where the event-based and periodic groups are shown as partial orders on the left. The figure on the right illustrates the possible merge options.

Options 1, 2, and 3 are all valid. However, *Option 4* is invalid because, according to RTLola's `event-based before periodic rule`, `y` must be evaluated after `c`, and `z` must be evaluated after `d`. This ordering is not respected in *Option 4*.

Among the valid options, *Option 3* results in the fewest evaluation levels. This implies earlier output and likely reduced memory usage, making it the most desirable choice.

Algorithm 5.7.1.2.1 (Heuristic algorithm to find the evaluation order)

```
FindEvalOrder(mir):
 1  (event_nodes, periodic_nodes) ⟵ SplitEventBasedAndPeriodic(mir)
 2  is_event_based ⟵ λ node: ¬node.is_periodic(mir)
 3  is_periodic ⟵ λ node: node.is_periodic(mir)
 4  event_roots ⟵ FindRoots(event_nodes, mir, is_event_based)
 5  periodic_roots ⟵ FindRoots(periodic_nodes, mir, is_periodic)
 6  orders_event ⟵ empty list
 7  for roots in event_roots do
 8      order ⟵ DagEvalOrder(roots, mir, is_event_based)
 9      orders_event.append(order)
10  orders_periodic ⟵ empty list
11  for roots in periodic_roots do
12      order ⟵ DagEvalOrder(roots, mir, is_periodic)
13      orders_periodic.append(order)
14  merged_event ⟵ MergeEvalOrdersByOffset(orders_event, mir, is_event_based)
15  merged_periodic ⟵ MergeEvalOrdersByOffset(orders_periodic, mir, is_periodic)
16  final_order ⟵ MergePeriodicAndEventBased(merged_event, merged_periodic, mir)
17  return final_order
```

In Algorithm 5.7.1.2.1, the input parameter `mir` refers to the RTLolaMIR intermediate representation produced by the RTLola frontend. It contains dependency information as well as all other details required for compilation.

Remark 5.7.1.2.2 (Reference to full algorithm). *The FindRoots algorithm is discussed in Appendix Appendix A-C. To keep the main text concise, we have omitted some of the other lower-level subroutines. However, the complete implementation — including the algorithm, end-to-end tests, evaluation, etc. — is available in the provided source code repository. Interested readers are encouraged to consult the source for full details.*

Chapter 6

Implementation of Architecture in Clash

This chapter presents the implementation of the architecture in Clash showing how various components have been realized. It assumes some familiarity with Haskell and Clash. Please refer to Section 3.3.2 in the background chapter to understand the main concepts used in this work.

6.1 Input and output streams

Streams in RTLola have static data types. For each stream, we define a custom data-type that wraps the stream's value along with a Bool flag indicating the presence of new data.

Example 6.1.1 (Example Inputs and Outputs)

```
data ValidInt = ValidInt {
    value :: Int,
    valid :: Bool
} deriving (Generic, NFDataX)

data Inputs = Inputs {
    input0 :: ValidInt,
    input1 :: ValidInt
} deriving (Generic, NFDataX)

data Outputs = Outputs {
    output0 :: ValidInt,
    output1 :: ValidInt,
    output2 :: ValidInt
} deriving (Generic, NFDataX)
```

Example 6.1.1 shows the data defined as record types (Example 3.3.2.8.3). The `Inputs` and `Outputs` is the collection of all inputs and outputs.

Furthermore, they derive `NFDataX` which is important in the type of data in Clash inorder for Clash to know that it can fully evaluate the data for synthesis.

6.2 Pacing type-system

Correct stream activation is crucial for accurate evaluation. To ensure this, we developed a type system for pacings. Despite the Clash monitor's code being generated by the compiler, which is tested with end-to-end tests, the pacing type system still enforces the correct usage of pacings according to RTLola semantics, providing an additional layer of security.

Let's consider an example to illustrate this better.

Example 6.2.1 (Example pacings)

```
input x : Int
input y : Int

output a @x := x + 1
output b @(x and y) := x + a
```

The example shows two output streams `a` and `b` with pacings `@x` and `@(x and y)` respectively. While RTLola frontend could automatically infer the pacing types here, it has been explicitly annotated for clarity.

The pacing of `@x` indicates that the stream `a` must be activated whenever there is an event in the input stream `x` and `@(x and y)` indicates the the stream must be activated only when both the input streams have the event in the cycle.

For the example, the compiler generates the pacing types as below.

Example 6.2.2 (Example Pacing types)

```
class Pacing a where getPacing :: a -> Bool

data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingIn1 = PacingIn1 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 PacingIn1
                    deriving (Generic, NFDataX)

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingIn1 where getPacing (PacingIn1 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1
    where getPacing (PacingOut1 x0 x1) = getPacing x0 && getPacing x1

data Pacings = Pacings {
    pacingIn0 :: PacingIn0,
    pacingIn1 :: PacingIn1,
    pacingOut0 :: PacingOut0,
    pacingOut1 :: PacingOut1
} deriving (Generic, NFDataX)
```

In Example 6.2.2, `PacingIn0` and `PacingIn1` are the pacings for input streams `x` and `y`, and `PacingOut0` and `PacingOut1` are the pacings for the output streams `a` and `b`. All of the

pacings implement the typeclass `Pacing` with the `getPacing` function. The `PacingOut0` consists of `PacingIn0` indicating the relationship `@x`. Similary, `PacingOut0` consists of `PacingIn0` and `PacingIn1`. The actual boolean value of pacing is obtained by the `getPacing` function which gives the result based on the logical relationship.

Similar to `Inputs` and `Outputs` all the pacings are packed into the `Pacings` type for ease.

The type-system enforces the type safety of pacing in places like stream evaluation. For example the function that evaluates output stream `b` would be typed with pacing like:

```
outputStream1 ::
  HiddenClockResetEnable dom => Signal dom PacingOut1 -> ...
outputStream1 ...
```

This guarantees that the pacing signal sent to `outputStream1` can only be the logical `AND` of pacings of the respective input streams.

6.3 High-level controller

As explained in Section 4.4.0.1, the high-level controller (HLC) is responsible for all timing and pacing control signals and handling the input data. It emits an event to the queue which holds information such as pacing signals for event-based and periodic streams, slide signals for sliding windows as well as the input data.

In Clash implementation, `Event` is basically a tuple holding all of those informations. Considering the specification has a sliding window, the compiler would generate `Event` alias type as `type Event = (Inputs, Slides, Pacings)`.

The pacing signals for event based streams depend on the logical combination of new data in streams. Any such logical dependency transitively depend on the input events. RTLola frontend [73] does this analysis and provides the event-based pacing based on only the input streams.

For periodic streams and sliding windows the time is tracked with the help internal timers. Such timers count the clock cycles and calculate time based on the known frequency of the clock.

Example 6.3.1 (Example specification to illustrate HLC)

```
input x : Int
input y : Int

output a @(x and y) := x + y
output b @1kHz := x.aggregate(over: 0.01s, using: sum) + y.hold(or: 0)
output c @100Hz := x.hold(or: 1) + b.hold(or: 0)
```

The Example 6.3.1, has event-based output stream `a` and the periodic ouput streams `b` and `c` along with the sliding window corresponding to the `aggregate` in output `b`.

The RTLola frontend [73] conviniently provides us the information about how many buckets the window should be divided in. This is done for efficient memory bound evaluation of sliding window. Consider Section 3.1.9 for more detail. In this particular case the window is divided into `10` buckets with the span of `0.001s` each. The sliding window forwards one bucket span each time, hence the sliding must happen every `0.001s`.

The periodic frequency of output streams indicate that the pacing signal for `b` and `c` must be generated every `0.001s` and `0.01s` respectively.

Example 6.3.2 (Generated HLC in Clash from the example specification)

```
hlc :: HiddenClockResetEnable dom
       => Signal dom Inputs
       -> Signal dom (Bool, Event)
hlc inputs = out
    where
        out = bundle (newEvent, event)
        newEvent = hasInput0 .||. hasInput1
                   .||. timer0Over .||. timer1Over

        event = bundle (inputs, slides, pacings)

        slides = Slides <$> s0
        pacings = Pacings <$> pIn0 <*> pIn1 <*> pOut0 <*> pOut1 <*> pOut2

        hasInput0 = ((.valid). (.input0)) <$> inputs
        hasInput1 = ((.valid). (.input1)) <$> inputs

        pIn0 = PacingIn0 <$> hasInput0
        pIn1 = PacingIn1 <$> hasInput1
        pOut0 = PacingOut0 <$> pIn0 <*> pIn1
        pOut1 = PacingOut1 <$> timer0Over
        pOut2 = PacingOut2 <$> timer1Over

        s0 = timer0Over

        timer0Over = timer0 .>=. period0InNs
        timer0 = timer timer0Over
        period0InNs = 1000000
        timer1Over = timer1 .>=. period1InNs
        timer1 = timer timer1Over
        period1InNs = 10000000

        timer :: HiddenClockResetEnable dom
                 => Signal dom Bool
                 -> Signal dom Int
        timer reset = register 0 (mux reset (pure deltaTime) nextTime)
            where
                nextTime = timer reset + pure deltaTime
                deltaTime = systemClockPeriodNs
```

The Example 6.3.2 shows that the HLC receives the input `Inputs` and outputs the `Event` along with a `Bool` indicating presence of new event. The expression for `newEvent` shows that whenever there is any input or timeouts, an event will be created.

The `timer` function shows how the timekeeping is done in HLC. On every clock cycle the value of time increase by the clock period. The `timer` resets the count on receiving the reset signal.

There are two call of `timer` functions i.e in `timer0` and `timer1`. Clash translates the function into a circuit component. Each call of function is an instantiation of the component. Notice how we only instantiate two timer components even though we have two periodic pacings and a slide to time. This is because the timer is shared between slide of the window and the pacing for output `b`, as they share the common period of `0.001s` i.e `1000000ns`. Each timer keeps counting the time until a threshold corresponding to the period is reached. When that happens, the timeover signals `timer0Over` and `timer1Over` resets the timer. The timeovers indicate the time to provide slide and pacing signals.

For event-based streams, we can see that the pacings of input streams depend on the existence of input data. The pacing for output stream `a` corresponds to the statement `pOut0 = PacingOut0 <$> pIn0 <*> pIn1`, indicating `a` depends on both `x` and `y`. The actual boolean signal for pacing is obtained by calling `getPacing` function as explained in Section 6.2.

6.4 Queue

The queue is responsible for ordering events with first-in-first-out scheme and buffering the events until consumed. As the HLC and LLC operate at their own pace, the `push` and `pop` actions in the queue can come at any cycle including both coming at the same time.

Example 6.4.1 (Queue state with input and output)

```
type QData = Event
type QMem = Vec QMemSize QData
type QCursor = Int
type QPush = Bool
type QPop = Bool
type QPushValid = Bool
type QPopValid = Bool

type QState = (QMem, QCursor)
type QInput = (QPush, QPop, QData)
type QOutput = (QPushValid, QPopValid, QData)
```

Example 6.4.1 shows the data structures in the queue. The buffer is built as a vector of `Event` type. For a finite hardware the size of the buffer must be known corresponding to the `QMemSize`. The `QCursor` is the internal pointer in queue indicating which position in the buffer the next event must go. The state of the queue `QState` holds the buffer along with the cursor. Queue receives the input of push and pop signals along with the actual event to store as seen by the `QInput`. Based on the input, queue changes it's state and output the data

is popped along with the feedback signal indicating if push and pop actions happened successfully. This feedback status signals are important to deal with cases such as buffer overflow/underflow in the queue.

Example 6.4.2 (Registers keeping state)

```
queue :: HiddenClockResetEnable dom
          => Signal dom QInput
          -> Signal dom QOutput
queue input = output
    where
        output = bundle (pushValid, popValid, outData)
        state = bundle (buffer, cursor)
        buffer = register (repeat nullEvent :: QMem) nextBufferSignal
        cursor = register 0 nextCursorSignal
        pushValid = register False nextPushValidSignal
        popValid = register False nextPopValidSignal
        outData = register nullEvent nextOutDataSignal

        nextBufferSignal = nextBuffer <$> buffer
                          <*> bundle (input, cursor)
        nextCursorSignal = nextCursor <$> cursor
                          <*> bundle (input, buffer)
        nextOutDataSignal = nextOutData
                          <$> bundle (input, cursor, buffer)
        nextPushValidSignal = nextPushValid
                            <$> bundle (input, cursor, buffer)
        nextPopValidSignal = nextPopValid <$> bundle (input, cursor)
```

Example 6.4.2 illustrates how the queue maintains its internal state using registers. Note the use of the applicative functor nature of Signal, which provides a clean and elegant way to lift regular functions into the Signal context. For more details on how this works, see Section 3.3.2.9.

Functions like `nextBuffer` and `nextCursor` accept the current state and input to determine the next state and output. Combined with the registers, these state transition functions effectively create Mealy machines.

Remark 6.4.1 (Clash mealy function). *Clash provides a high-level `mealy` function to simplify this process, eliminating the need for manual register maintenance (see Section 3.3.2.10 for detail). However, we opted for an explicit implementation of our own Mealy machines for finer control.*

Example 6.4.3 (State transition functions)

```
nextBuffer :: QMem -> (QInput, QCursor) -> QMem
nextBuffer buf ((push, pop, qData), cur) = out
    where
        out = case (push, pop) of
            (True, True) -> qData +>> buf
            (True, False) -> if cur == length buf
                             then buf else qData +>> buf
            (False, _) -> buf

nextCursor :: QCursor -> (QInput, QMem) -> QCursor
nextCursor cur ((push, pop, _), buf) = out
    where
        out = case (push, pop) of
            (True, False) -> if cur == length buf then cur else cur + 1
            (False, True) -> if cur == 0 then 0 else cur - 1
            (_, _) -> cur

nextOutData :: (QInput, QCursor, QMem) -> QData
nextOutData ((push, pop, qData), cur, buf) = out
    where
        out = case (push, pop) of
            (True, True) -> if cur == 0 then qData else buf !! (cur - 1)
            (False, True) -> if cur == 0
                             then nullEvent else buf !! (cur - 1)
            (_, _) -> nullEvent

nextPushValid :: (QInput, QCursor, QMem) -> QPush
nextPushValid ((push, pop, _), cur, buf) = out
    where
        out = case (push, pop) of
            (True, True) -> True
            (True, False) -> cur /= length buf
            (False, _) -> False

nextPopValid :: (QInput, QCursor) -> QPop
nextPopValid ((push, pop, _), cur) = out
    where
        out = case (push, pop) of
            (True, True) -> True
            (False, True) -> cur /= 0
            (_, False) -> False
```

The transition functions in Example 6.4.3 implement the core logic of the queue. For instance, the `nextBuffer` function illustrates how the queue buffer is updated based on the presence of `push` and `pop` instructions. When a push occurs, the new data is inserted at the front of the buffer using the `+>>` function from Vector in Clash. This function shifts the existing elements one position to the right, making room at the front. As a result, the item at the far end of the buffer is discarded.

Once the queue signals a successful push using the `QPushValid` signal, the inserted data is retained in the buffer until it is consumed. To prevent data loss and ensure stable evaluation, the push operations are rejected if the queue is already full.

The current cursor position is tracked using `nextCursor`. Similarly, the functions `nextPushValid`, `nextPopValid`, and `nextOutData` manage the control and output data signals. Note that `!!` is a Clash vector function used to access elements by index.

6.5 Low-level controller

The Low-Level Controller (LLC) pops events from the queue and evaluates them to generate outputs for the monitor. Its design is guided by the dependency graph analysis (Section 5), which determines the evaluation order, pipelining strategy using `pipeline_wait`, and memory requirements. For more details on the architecture, refer to Section 4.4.0.3.

To better understand how the different components of the LLC are implemented in Clash, let's walk through an example.

Example 6.5.1 (Example RTLola specification to illustrate LLC)

```
input x : Int

output a := x + b.offset(by: -1).defaults(to:
                x.offset(by: -1).defaults(to: 0)
            )
output b := c.hold(or: a.offset(by: -1).defaults(to: 0)) - a
output c @1kHz := b.aggregate(over: 0.1s, using: sum)
                  + a.aggregate(over: 0.01s, using: sum)
```

The RTLola specification in Example 6.5.1 defines two event-based output streams, `a` and `b`, and one periodic output stream, `c`. Stream `a` depends on past values of `b` and `x`, with a default value nested inside another default. RTLola allows such default chains to be nested arbitrarily deep. Similarly, stream `b` depends on both `a` and `c`, while `c` depends on `b` and `a`. Since `b` and `c` operate asynchronously, the most recent value of `c` is accessed using the `hold` operator. If no previous value of `c` is available, the offset value of `a` is used instead — which itself may fall back on its own nested default.

From the specification, the graph analysis gives us:

1. Evaluation order: $\{x\} \rightarrow \{a\} \rightarrow \{b\} \rightarrow \{\text{sw(b,c)}, \text{sw(a,c)}\} \rightarrow \{c\}$
2. `pipeline_wait`: 2
3. Required memory for: `x = 2`, `a = 2`, `b = 1`, `sw(b,c) = 1`, `sw(a,c) = 1`, `c = 1`

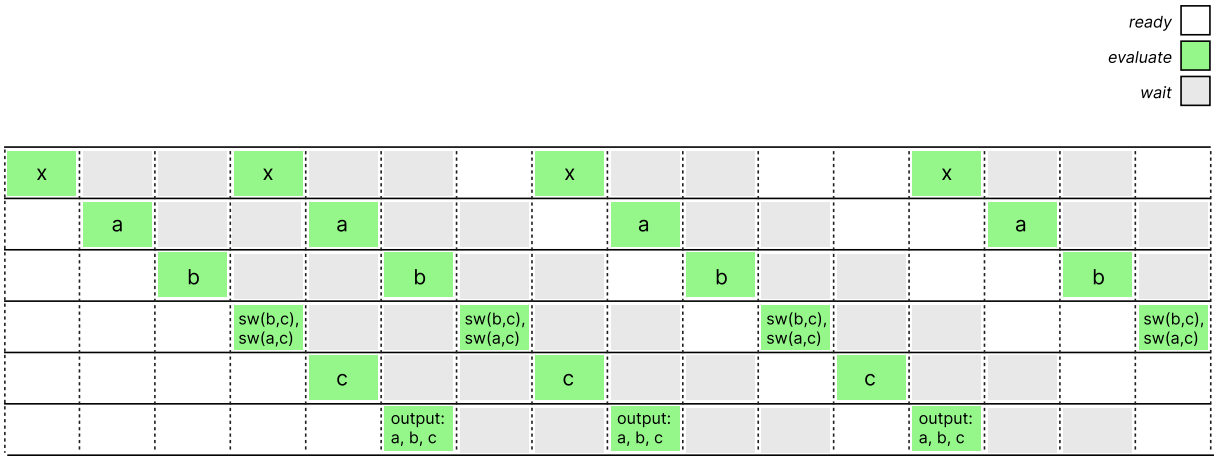


Figure 43: Visualiztion of pipeline for the example specification

Figure Figure 43 illustrates a possible execution trace of the example specification by the LLC. In the figure, columns correspond to clock cycles, while rows represent pipeline stages. The first two evaluations occur back-to-back. According to the `pipeline_wait` value of `2`, the system must wait two clock cycles before initiating the next evaluation. However, evaluation only proceeds if there is an event to process. The delayed start of the third and fourth evaluations in the figure demonstrates cycles where no events were available.

6.5.1 Control and data signals for evaluation

Example 6.5.1.1 (Generated LLC code for controlling the evalution)

```
llc :: HiddenClockResetEnable dom
       => Signal dom (Bool, Event)
       -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        input0 = (.input0) <$> inputs

        slide0 = (.slide0) <$> slides
        slide1 = (.slide1) <$> slides

        pIn0 = (.pacingIn0) <$> pacings
        pOut0 = (.pacingOut0) <$> pacings
        pOut1 = (.pacingOut1) <$> pacings
        pOut2 = (.pacingOut2) <$> pacings

        enIn0 = delayFor d1 nullPacingIn0 pIn0
        enOut0 = delayFor d2 nullPacingOut0 pOut0
        enOut1 = delayFor d3 nullPacingOut1 pOut1
        enSw1 = delayFor d4 nullPacingOut0 pOut0
        sld1 = delayFor d4 False slide1
        enSw0 = delayFor d4 nullPacingOut1 pOut1
        sld0 = delayFor d4 False slide0
        enOut2 = delayFor d5 nullPacingOut2 pOut2

        output0Aktv = delayFor d6 False (getPacing <$> pOut0)
        output1Aktv = delayFor d6 False (getPacing <$> pOut1)
        output2Aktv = delayFor d6 False (getPacing <$> pOut2)

        ...
```

Example 6.5.1.1 shows the generated code responsible for pacing and initiating the evaluation, based on the specification in Example 6.5.1. The presence of an event is indicated by a boolean flag received by the LLC. When ready, as indicated by `isPipelineReady`, the LLC sends the pop signal to the queue. The pipelined evaluation starts — signaled by `startNewPipeline` — when the LLC is ready and a valid event is available. The `pipelineReady` component keeps track of the wait time according to the specified `pipeline_wait`.

Example 6.5.1.2 (Ready after waiting pipeline_wait)

```
pipelineReady :: HiddenClockResetEnable dom =>
                    Signal dom Bool -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 2 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                    (mux (toWait .>. pure 0) (toWait - 1) toWait)
```

Example 6.5.1.2 illustrates the mechanism for waiting after starting a new pipeline evaluation. In this example, the `waitTime` is set to `2`. When `pipelineReady` receives a reset signal indicating the start of a new pipeline, it begins counting down from `waitTime` to zero. The pipeline is considered ready once the countdown reaches zero. The code for `pipelineReady` is only generated if needed i.e when `pipeline_wait > 0`.

From each event popped from the queue, the inputs, slide information, and pacing data are extracted. The pacing data is used to generate enable signals for evaluation steps. Notably, the `delayFor` function (shown in Example 6.5.1.1) is used with parameters like `d1`, `d2`, etc., to introduce delays. This custom function delays a signal by a specified number of clock cycles, taking a type-level natural number as its parameter. The values `d1`, `d2`, and so on are type-level representations of the natural numbers `1`, `2`, etc. Since these values are known at compile time, the Clash compiler can accurately compute the required circuit size and timing behavior.

These delays ensure that individual streams and the slide signals for sliding windows are activated in the correct sequence, adhering to the evaluation order. In this example, the enable signals are triggered in the following order:
$\text{enIn0} \rightarrow \text{enOut0} \rightarrow \text{enOut1} \rightarrow \text{enSw1}, \text{enSw0} \rightarrow \text{enOut2}$
This corresponds to the evaluation order:
$\{x\} \rightarrow \{a\} \rightarrow \{b\} \rightarrow \{\text{sw(b,c)}, \text{sw(a,c)}\} \rightarrow \{c\}.$

Example 6.5.1.3 (delayFor function)

```
delayFor :: forall dom n a .
            (HiddenClockResetEnable dom, KnownNat n, NFDataX a)
              => SNat n -> a -> Signal dom a -> Signal dom a
delayFor n initVal sig = last delayedVec
    where
      delayedVec :: Vec (n + 1) (Signal dom a)
      delayedVec = iterateI (delay initVal) sig
```

Example 6.5.1.3 shows the definition of the `delayFor` function. The delay function in Clash delays a `Signal` by one clock cycle, using a given default value for time `0`. The `iterateI` function takes a function `f` and a starting value `x`, and returns a `Vec` beginning with `x`, followed by `n` repeated applications of `f`, where `n` is inferred from the context. By combining

these high-level constructs, we build a chain of delay stages applied one after another to achieve the desired number of delay cycles.

6.5.2 Tags

As explained in Section 4.4.0.3, we annotate a result with a tag for easy access from a window of values.

Example 6.5.2.1 (Tags)

```
...

tIn0 = genTag (getPacing <$> pIn0)
tOut0 = genTag (getPacing <$> pOut0)
tOut1 = genTag (getPacing <$> pOut1)
tSw1 = genTag (getPacing <$> pOut0)
tSw0 = genTag (getPacing <$> pOut1)
tOut2 = genTag (getPacing <$> pOut2)

-- tag generation takes 1 cycle so we need to delay the input data
input0Data = delay 0 (((.value). (.input0)) <$> inputs)

-- delayed tags to be used in different levels
tagsDefault = Tags nullT nullT nullT nullT nullT nullT
curTags = Tags <$> tIn0 <*> tOut0 <*> tOut1
               <*> tOut2 <*> tSw0 <*> tSw1
curTagsLevel1 = delayFor d1 tagsDefault curTags
curTagsLevel2 = delayFor d2 tagsDefault curTags
curTagsLevel3 = delayFor d3 tagsDefault curTags
curTagsLevel4 = delayFor d4 tagsDefault curTags
curTagsLevel5 = delayFor d5 tagsDefault curTags
nullT = invalidTag

genTag :: HiddenClockResetEnable dom
          => Signal dom Bool -> Signal dom Tag
genTag en = t
    where
        t = register 1 (mux en next_t t)
        next_t = mux (t .==. (pure maxTag)) (pure 1) (t + 1)

...
```

Example 6.5.2.1 shows the code segment from the example responsible for generating tags for the streams. These tags are used to annotate the results produced in the current evaluation cycle. Since each stream can be activated at different times, we assign individual tags to each — such as `tIn0`, `tOut0`, etc.

Additionally, we keep track of the tags associated with each level or position in the evaluation order. This is done by simply delaying the tags by the appropriate amount for each level. This mechanism is crucial, as each evaluation cycle can correspond to a different set of tags.

The tag generation function `genTag` itself just counts upward from `1` to a maximum value and then restarts the count. The maximum value is chosen to ensure that no duplicate tags can appear within any window of results.

Example 6.5.2.2 (Matching tags)

```
getMatchingTag :: KnownNat n => Vec n (Tag, a) -> Tag -> a -> a
getMatchingTag win tag dflt = out
    where
        out = case findIndex (\(t, _) -> t == tag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getMatchingTagFromNonVec :: (Tag, a) -> Tag -> a -> a
getMatchingTagFromNonVec (tag, dta) tagToMatch dflt =
    if tag == tagToMatch then dta else dflt
```

Example 6.5.2.2 shows functions to find values with a matching tag. The `getMatchingTag` function takes in a window of tagged value along with a tag to match and returns the matching value if found, otherwise returns the provided default value. The `getMatchingTagFromNonVec` functions works similary for a single tagged data.

6.5.3 Offset access

The value from a past offset is access with the help of tag.

Example 6.5.3.1 (Offset access)

```
getOffset :: KnownNat n => Vec n (Tag, a) -> Tag -> Tag -> a -> a
getOffset win tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = case findIndex (\(t, _) -> t == offsetTag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getOffsetFromNonVec :: (Tag, a) -> Tag -> Tag -> a -> a
getOffsetFromNonVec (winTag, winData) tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = if offsetTag == winTag then winData else dflt

earlierTag :: Tag -> Tag -> Tag
earlierTag curTag cyclesBefore = if curTag > cyclesBefore
    then curTag - cyclesBefore else curTag - cyclesBefore + maxTag
```

As shown in Example 6.5.3.1, the `getOffset` and `getOffsetFromNonVec` functions are used to access values from a sliding window of tagged values and a single tagged value, respectively. To retrieve the correct value based on an offset, the `earlierTag` function is used to compute the appropriate tag, accounting for tag overflow. The value is then retrieved by matching against this earlier tag. In the case of a window of values, the

`findIndex` function is used, a high-level Clash function that returns the index of a vector element satisfying a given predicate.

6.5.4 Hold access

For the `hold` access it is a simple matter of returning the latest value.

Example 6.5.4.1 (Hold access)

```
getLatestValue :: KnownNat n => Vec (n + 1) (Tag, a) -> a -> a
getLatestValue win dflt =
    let (tag, dta) = last win
    in if tag == invalidTag then dflt else dta

getLatestValueFromNonVec :: (Tag, a) -> a -> a
getLatestValueFromNonVec (tag, dta) dflt =
    if tag == invalidTag then dflt else dta
```

6.5.5 Input stream

Example 6.5.5.1 (Input stream)

```
input0Window :: HiddenClockResetEnable dom
    => Signal dom PacingIn0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
input0Window en tag val = result
    where result = register (repeat (invalidTag, 0))
                            (mux (getPacing <$> en)
                              ((<<+) <$> result <*> (bundle (tag, val)))
                              result)
```

Example 6.5.5.1 shows the input window corresponding to the input stream. Note the required memory of `2` in input `x` mentioned in Example 6.5.1. As the new pipeline cycle can start with a new value of `x` already before the old value was fully consumed, it is important to keep a window of `2` results.

The `input0Window` is enabled with the pacing signal `PacingIn0`. When the pacing signal is active, the input value gets tagged and put into the window. The `<<+` function in Clash adds and element to the tail of the vector shifting the existing elements by one step to the left. The leftmost value gets overflowen.

6.5.6 Evaluation of event-based streams

Example 6.5.6.1 (Evaluation of event-based streams)

```
outputStream0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
outputStream0 en tag in0_0 out1_1 = result
    where
        result = register (repeat (invalidTag, 0))
                    (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = in0_0 + out1_1


outputStream1 :: HiddenClockResetEnable dom
    => Signal dom PacingOut1
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom (Tag, Int)
outputStream1 en tag out2_0 out0_1 = result
    where
        result = register (invalidTag, 0)
                  (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = out2_0 - out0_1
```

Example 6.5.6.1 shows how output streams `a` i.e `outputStream0` and `b` i.e `outputStream1` are calculated. The streams receives the pacing signal, tag and the depending data as inputs. It calculates the results when activated by pacing signal as per the expression and returns the result. In the case of `a` the data are simply added whereas for `b` they are substracted. The actual evaluation of the expression of the dependencis happens elsewhere and the resulting values are provided here.

6.5.7 Periodic stream and sliding windows

Example 6.5.7.1 (Period stream and sliding windows)

```
outputStream2 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Tag
    -> Signal dom (Vec 101 Int)
    -> Signal dom (Vec 11 Int)
    -> Signal dom (Tag, Int)
outputStream2 en tag sw0 sw1 = result
    where
        result = register (invalidTag, 0)
                  (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (merge0 <$> sw0) + (merge1 <$> sw1)
        merge1 :: Vec 11 Int -> Int
        merge1 win = fold windowBucketFunc1 (tail win)
        merge0 :: Vec 101 Int -> Int
        merge0 win = fold windowBucketFunc0 (tail win)

windowBucketFunc0 :: Int -> Int -> Int
windowBucketFunc0 acc item = acc + item

slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut1
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 101 Int))
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag,
                       nextWindow <$> (snd <$> window)
                       <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 101 Int

        nextWindow :: Vec 101 Int
            -> Bool -> Bool -> Int -> Vec 101 Int
        nextWindow win toSlide newData dta = out
            where
                out = case (toSlide, newData) of
                    (False, False) -> win
                    (False, True) -> lastBucketUpdated
                    (True, False) -> 0 +>> win
                    (True, True) -> 0 +>> lastBucketUpdated
                lastBucketUpdated = replace 0
                            (windowBucketFunc0 (head win) dta) win
```

Example 6.5.7.1 shows how the periodic output with sliding windows are calculated. The `outputStream2` corresponds to the stream `c` in the example. The stream `c` depends on the

two sliding window aggregates from `b` and `a`. It then adds them together to get the periodic output in `c`. The `outputStream2` has the usual inputs of pacing and tag, then it takes in two vectors i.e `Signal dom (Vec 101 Int)` and `Signal dom (Vec 11 Int)`, corresponding to the sliding windows. As the sliding windows are so similar, in Example 6.5.7.1, only one sliding window is shown.

The `outputStream2` merges the two sliding windows with `merge0 <$> sw0` and `merge1 <$> sw1`. It then adds them together as needed by the expression. Merge happens with the `fold` function in Clash, which reduces the vector by merging in tree-like fashion by applying the binary function again and again. In this particular case the function `windowBucketFunc0` basically is a sum function corresponding to the `sum` in `aggregate` expression.

The `slidingWindow0` is one of the function maintaining the sliding window. On top of pacing signal, tag and data input, it also receives an extra boolean input `slide` which controls when the window should slide. Remember the slide signal comes from the HLC similar to the pacings. Compared to other streams, the sliding window must be activated when there are any new events as well as when it is time to slide. The `en = (getPacing <$> newData) .||. slide` ensures that. The `nextWindow` function holds the actual logic for the sliding window. The `lastBucketUpdated` shows how the partial-aggregation happens for a new data. The pattern-matching part shows how the sliding window is updated in various conditions.

6.5.8 Running evaluations with dependencies

Example 6.5.8.1 (Wiring for evaluation)

```
    ...
    -- Evaluation of input windows: level 0
    input0Win = input0Window enIn0 tIn0 input0Data

    -- Evaluation of output 0: level 1
    out0 = outputStream0 enOut0
            ((.output0) <$> curTagsLevel1)
            out0Data0 out0Data1
    ...

    -- Evaluation of output 1: level 2
    out1 = outputStream1 enOut1
            ((.output1) <$> curTagsLevel2)
            out1Data0 out1Data1
    ...

    -- Evaluation of sliding window 0: level 3
    sw0 = slidingWindow0 enSw0 sld0 ((.slide0) <$> curTagsLevel3) sw0Data
    (_, sw0Data) = unbundle out1

    -- Evaluation of sliding window 1: level 3
    sw1 = slidingWindow1 enSw1 sld1 ((.slide1) <$> curTagsLevel3) sw1Data
    sw1Data = getMatchingTag <$> out0
              <*> ((.output0) <$> curTagsLevel3) <*> (pure 0)

    -- Evaluation of output 2: level 4
    out2 = outputStream2 enOut2 ((.output2) <$> curTagsLevel4)
            out2Data0 out2Data1
    (_, out2Data0) = unbundle sw0
    (_, out2Data1) = unbundle sw1

    -- Outputing all results: level 5
    output0 = ValidInt <$> output0Data <*> output0Aktv
    output0Data = getMatchingTag <$> out0
                  <*> ((.output0) <$> curTagsLevel5)
                  <*> (pure 0)
    output1 = ValidInt <$> output1Data <*> output1Aktv
    (_, output1Data) = unbundle out1
    output2 = ValidInt <$> output2Data <*> output2Aktv
    (_, output2Data) = unbundle out2

    outputs = Outputs <$> output0 <*> output1 <*> output2

    ...
```

Example 6.5.8.1 shows how the evaluation happens for the example. Evaluation is basically a respective function calls wiring the right control and data signals.

6.5.9 Recursive nesting of defaults

Let's consider a more complicated example to understand how the recursive nesting of defaults is handled.

Example 6.5.9.1 (Recursive nesting of defaults)

```
input x : Int
input y : Int

output a := y + x.offset(by: -2).defaults(to:
                  y.offset(by: -1).defaults(to: 10)
                  + a.offset(by: -1).defaults(to:
                    x.offset(by: -1).defaults(to: 20)))
```

Example 6.5.9.2 (Handling recursive nesting of defaults)

```
...

out0 = outputStream0 enOut0 ((.output0) <$> curTagsLevel1)
        out0Data0 out0Data1
out0Data0 = getMatchingTag <$> input1Win
            <*> ((.input1) <$> curTagsLevel1)
            <*> (pure (0))
out0Data1 = getOffset <$> input0Win
            <*> ((.input0) <$> curTagsLevel1)
            <*> (pure 2) <*> out0Data1Dflt
out0Data1Dflt = out0Data1DfltData0 + out0Data1DfltData1
out0Data1DfltData0 = getOffset <$> input1Win
                     <*> ((.input1) <$> curTagsLevel1)
                     <*> (pure (1)) <*> (pure (10))
out0Data1DfltData1 = getOffsetFromNonVec <$> out0
                     <*> ((.output0) <$> curTagsLevel1)
                     <*> (pure (1)) <*> out0Data1DfltData1Dflt
out0Data1DfltData1Dflt = getOffset <$> input0Win
                     <*> ((.input0) <$> curTagsLevel1)
                     <*> (pure (1)) <*> (pure (20))

...
```

Example 6.5.9.2 shows the exact call of the main expression of `a` i.e `outputStream0` with the dependencies. The access to the offset of `x` happens in `out0Data1`, which when not availabe defaults to `out0Data1Dflt`. The `out0Data1Dflt` inturn is an expression with a sum of two subexpressions of offsets. One of the subexpression again has the the inner subexpression. This goes on an on.

Evaluation in this case is a recursive process of unrolling the expression. It can be thought of as a linearization of tree to produce a sequence of statements.

6.6 Connecting all components together

Example 6.6.1 (Monitor)

```
monitor :: HiddenClockResetEnable dom
           => Signal dom Inputs
           -> Signal dom Outputs
monitor inputs = outputs
    where
        (newEvent, event) = unbundle (hlc inputs)

        (qPushValid, qPopValid, qPopData) =
            unbundle (queue (bundle (qPush, qPop, qInptData)))
        qPush = newEvent
        qPop = toPop
        qInptData = event

        (toPop, outputs) = unbundle (llc (bundle (qPopValid, qPopData)))
```

Example 6.6.1 shows how all the individual components of the monitor fit together. The `hlc` takes in an inputs and produces `newEvent` and `event`. The `newEvent` implies the push i.e `qPush` signal to the queue, and `event` corresponds to the input data to the queue. The pop signal i.e `qPop` of the queue is wired to the `llc`, along with `qPopValid` and `qPopData`. The `llc` produces the `qPop` signal when it wants to pop the data which will be availabe in `qPopData`. The result from the `llc` comes in `outputs` which is wired outside the monitor as the result.

6.7 Remarks

This capter showed how various parts of the architecture have been implemented in Clash. Please refer to the appendix to see an example of complete code in Clash generated by our compiler.

As seen from the code snippets in this chapter, Clash provided a highly expressive and powerful way to build the architecture. It allowed us to define our own type systems, which helped enforce RTLola semantics, such as through the pacing type system. Clash made it easy to express parametric hardware constructs, like with the `delayFor` function. Pattern matching was especially useful in writing clean and readable transition functions, such as in the queue implementation. High-level functions like `iterateI` and `fold` enabled concise expression of complex operations. Additionally, the `applicative functor` nature of `Signal` helped separate logic from wiring, making the design both modular and flexible.

Chapter 7

Evaluation

This chapter presents the evaluation we performed to analyze the impact of pipeline evaluation. It also compares our compiler with the existing compiler by examining the synthesized circuit primitives of the resulting hardware monitors.

7.0.1 Specifications used in evaluation

For the evaluation, we collected a range of RTLola specifications. Some were drawn from examples in prior work on monitoring for unmanned aerial vehicles [37], [20], while others were handwritten to represent moderately complex specifications with diverse stream interactions.

Since our compiler does not yet support the full RTLola language, such as mathematical functions, many existing examples could not be used directly. Additionally, several of these examples rely on outdated syntax. The existing hardware monitor compiler [6] also supports only a limited subset of RTLola features. As a result, we adapted the selected examples to ensure compatibility with both compilers while preserving their original semantics as much as possible.

To keep the presentation in this chapter clean and readable, the full specifications have been placed in the appendix. Stream names in the original specifications can be long and difficult to display clearly in tables or figures. Therefore, we use shortened nicknames: `i0`, `i1`, etc., for input streams; `sw0`, `sw1`, etc., for sliding windows; and `o0`, `o1`, etc., for output streams, based on the order in which they are defined. These nicknames will be used consistently throughout this chapter.

The following table provides a reference linking each specification to its location in the appendix. It also lists the number of evaluation nodes (streams and sliding windows), the number of sliding window buckets, and the corresponding `pipeline_wait` value.

Spec	Location	Evaluation nodes	Sliding window buckets	pipeline_wait
Spec1	Appendix A-BA	21	215	0
Spec2	Appendix A-BB	7	10	0
Spec3	Appendix A-BC	8	8	0
Spec4	Appendix A-BD	9	0	2
Spec5	Appendix A-BE	5	0	2
Spec6	Appendix A-BF	5	10	0
Spec7	Appendix A-BG	5	0	0
Spec8	Appendix A-BH	5	0	1
Spec9	Appendix A-BI	7	200	2

Table 1: Specifications used in evaluation

7.0.2 Comparison of evaluation order

The table below presents the evaluation order derived from our graph analysis, which is used during evaluation in the LLC stage of our architecture. For comparison, it also shows the stream evaluation order used in the LLC of the existing compiler. Since the existing compiler evaluates event-based and periodic streams alternately, we separate them in the table for clarity.

Note that in the existing architecture, the first evaluation step is always dedicated to input preparation. As a result, the periodic evaluation order does not perform any computation during this initial step.

RTLola spec	Order of evaluation in our architecture	Order of evaluation in existing architecture
Spec1	1. i0, i2, i1 2. o0 3. o2, o1 4. o3 5. sw0 6. o4, sw4, sw5, sw3, sw2, sw1, sw6 7. o5, o8, o7, o6, o9, o10	*Event-based*: 1. i0, i1, i2 *Periodic*: 1. - 2. sw1, sw2, sw3, sw4, sw5, sw6 3. o0, o6, o7, o8, o9, o10 4. o1, o2 5. o3 6. sw0 7. o4 8. o5
Spec2	1. i1, i0 2. o0 3. o1 4. o2 5. sw0 6. o3	*Event-based*: 1. i0, i1 2. o0 3. o1 4. o2

RTLola spec	Order of evaluation in our architecture	Order of evaluation in existing architecture
		Periodic: 1. - 2. sw0 3. o3
Spec3	1. i1, i2, i0 2. o1 3. sw0, sw1 4. o0, o2	*Event-based*: 1. i0, i1, i2 2. o1 *Periodic*: 1. - 2. sw0, sw1 3. o0, o2
Spec4	1. i0, i1 2. o0, o3, o4 3. o5, o1 4. o6, o2	*Event-based*: 1. i0, i1 2. o0, o3 *Periodic*: 1. - 2. o1, o4 3. o2, o5 4. o6
Spec5	1. i0 2. o0 3. o1, o3 4. o2	*Event-based*: 1. i0 2. o0 3. o1 4. o2 *Periodic*: 1. - 2. o3
Spec6	1. i1, i0 2. o0 3. sw0 4. o1	*Event-based*: 1. i0, i1 2. o0 *Periodic*: 1. - 2. sw0 3. o1
Spec7	1. i0, o0, o1, o2 2. o3	*Event-based*: 1. i0 2. o0, o1, o2

RTLola spec	Order of evaluation in our architecture	Order of evaluation in existing architecture
		Periodic: 1. - 2. o3
Spec8	1. i0, o0, o1 2. o2 3. o3	*Event-based*: 1. i0 2. o0, o1 3. o2 *Periodic*: 1. - 2. o3
Spec9	1. i0, o0 2. o1 3. sw0 4. o2 5. sw1 6. o3	*Event-based*: 1. i0 2. o0 3. o1 *Periodic*: 1. - 2. sw0 3. o2 4. sw1 5. o3

A side-by-side comparison of the evaluation orders shows that the orders in our approach can differ significantly from those of the existing compiler. For instance, in *Spec7* and *Spec8*, output nodes are evaluated together with input nodes. This is a result of offset-related optimizations (see Section 5.5). Additionally, the heuristic-based merging strategy (see Section 5.7) can lead to noticeably different orders, as seen in *Spec1*.

7.0.3 Comparison of evaluation throughput

As defined for pipeline throughput in Definition 4.3.4, the evaluation throughput refers to the number of evaluation cycles completed in a single clock cycle when the system is saturated with inputs and operating in a steady-state condition. The concept of steady-state is important in pipeline evaluation because while the initial output incurs some latency, subsequent outputs can be produced continuously once the pipeline is filled.

By saturated inputs, we mean that the monitor receives sufficient stream of inputs, ensuring that it never has to wait to start the next evaluation cycle.

Remark 7.0.3.1 (Ignoring queue fetch latency). *Between successive evaluations, the LLC must fetch data from the queue, which can take some cycles depending on the implementation. For simplicity, we have ignored this overhead in order to focus on the fundamental differences in the evaluation mechanisms.*

To better understand the comparison, let's examine the evaluation of *Spec8*. According to Table 1, the pipeline_wait for *Spec8* is `1`, indicating that evaluation in the LLC follows a pipelined fashion, as illustrated in Figure 44.

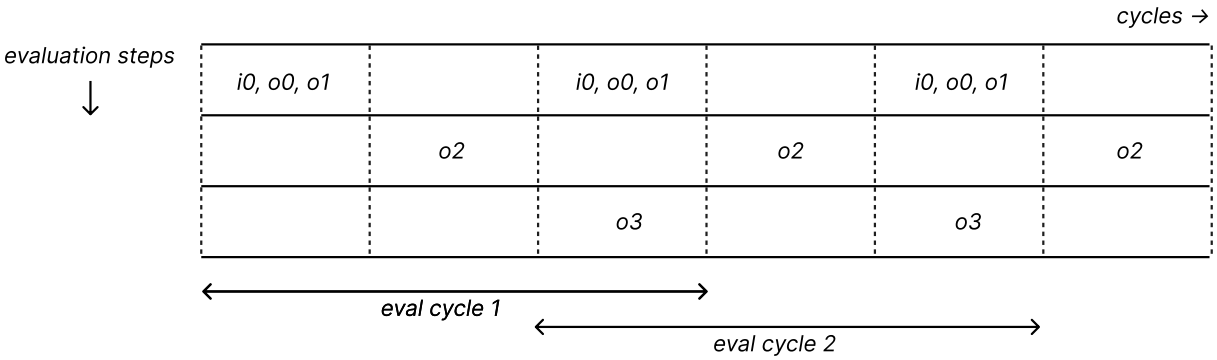


Figure 44: Evaluation of Spec8 in LLC by our architecture

In this setup, when the system is saturated with inputs, a complete evaluation can be completed every two cycles, resulting in a throughput of $\frac{1}{2}$.

In contrast, the existing architecture (shown in Figure 45) evaluates event-based and periodic streams alternately. It requires five cycles to complete one full evaluation before the next cycle can begin, yielding a throughput of $\frac{1}{5}$.

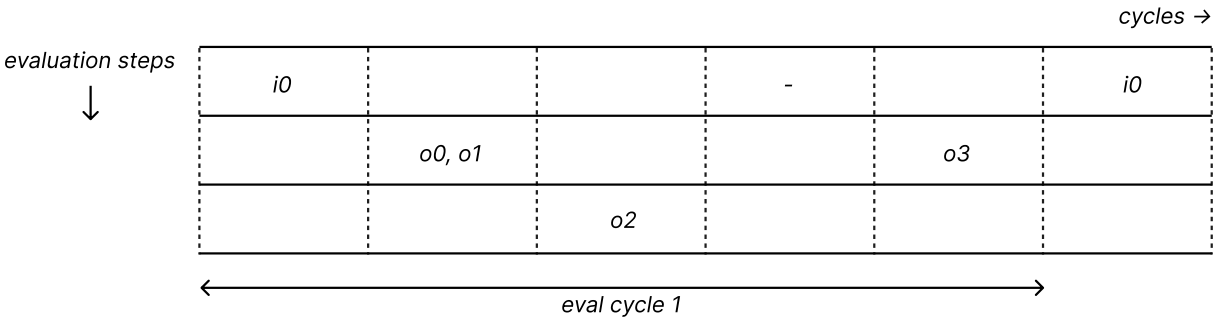


Figure 45: Evaluation of Spec8 in LLC by the existing architecture

Comparing evaluation throughput for other specs in similar way, we get the following result:

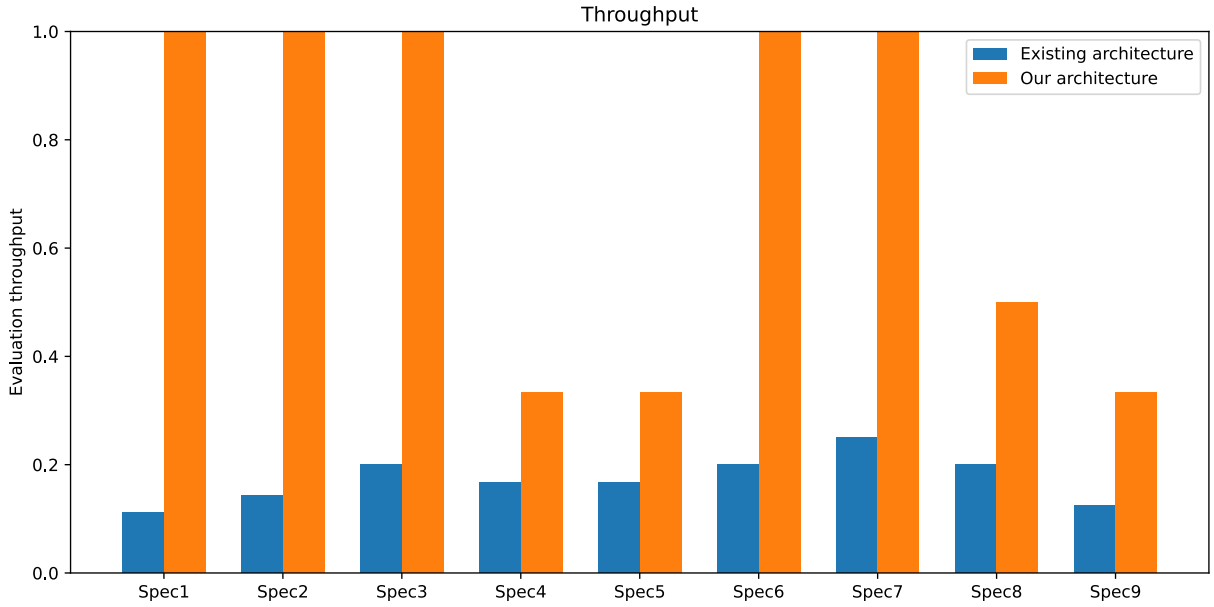


Figure 46: Evaluation throughput compared with existing architecture

Figure 46 shows a significant improvement in evaluation throughput due to pipelining. Even when a `pipeline_wait` is present, the pipelined evaluation consistently outperforms the existing architecture. Since the maximum wait is bounded by the length of the evaluation order, the worst-case throughput is comparable to that of the existing design, and never worse.

Remark 7.0.3.2 (Pipeline evaluation potential in real-world specifications). *The benefits of pipeline evaluation become more significant as specifications grow in complexity with the length of evaluation order. Notably, most UAV-related specifications we examined had small* pipeline_wait *values, even though they were fairly large. This suggests that pipelining might offer substantial performance gains in many real-world scenarios, though a more comprehensive study is needed to fully validate this.*

7.0.4 Comparision of synthesized hardware primitives

To evaluate the generated hardware monitor from our Clash-based implementation, we compared the VHDL generated by the existing compiler with the VHDL produced after compiling the generated Clash code. For a fair comparison, both VHDL designs were synthesized.

We used open-source tools: GHDL to compile the VHDL code and Yosys [49], along with its GHDL plugin, to synthesize the design into an RTL netlist. During synthesis, Yosys performs a series of cleanup and optimization passes. From the resulting netlist, we obtained statistics on various circuit primitives such as logic gates, flip-flops, and multiplexers. This netlist is general and not optimized or mapped to the resources of any specific device.

To evaluate the generated hardware monitor with our Clash based implementation we compared the generated VHDL from the existing compiler with the generated VHDL

produced after compiling the generated Clash code. For fair comparision we synthesized the generated VHDLs.

For this, we used open-source tools: GHDL to compile the VHDL code and, Yosys [49] along with its GHDL plugin, to synthesize the design into an RTL netlist. During synthesis, Yosys applies a series of cleanup and optimization passes. From the resulting netlist, we obtain statistics on various circuit primitives such as logic gates, flip-flops, and multiplexers. This is a general netlist without any optimization or mapping to the resources of a specific target device / ASIC.

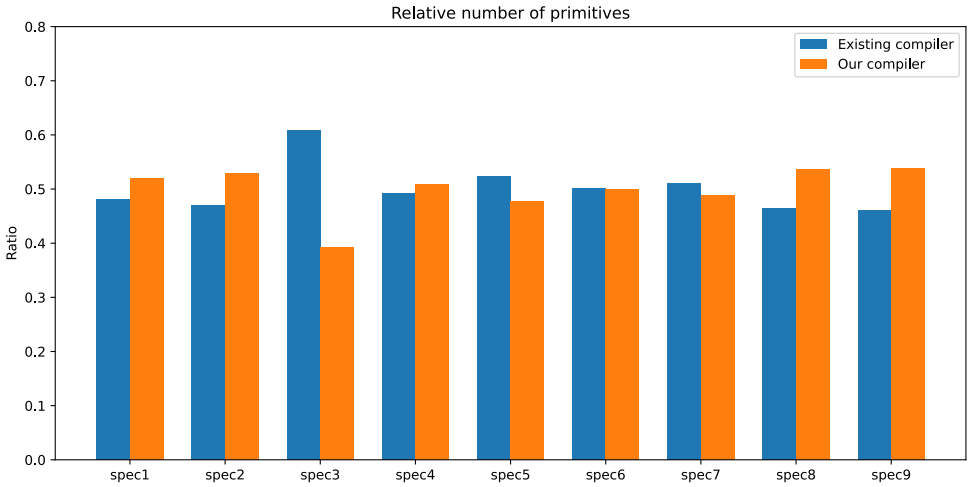


Figure 47: Relative number of synthesized primitives

Figure 47 shows the relative number of synthesized primitives between the two netlists for each evaluated RTLola specification. The count includes all primitives, such as logic gates, flip-flops, and multiplexers, but excludes wires. The ratios in the figure are computed by dividing the number of primitives from one compiler by the total number from both compilers combined. This gives us a relative view for comparision. The result however, doesn't show any significant difference.

However, when we categorically look at the sequential and combinational primitives we see a clear difference.

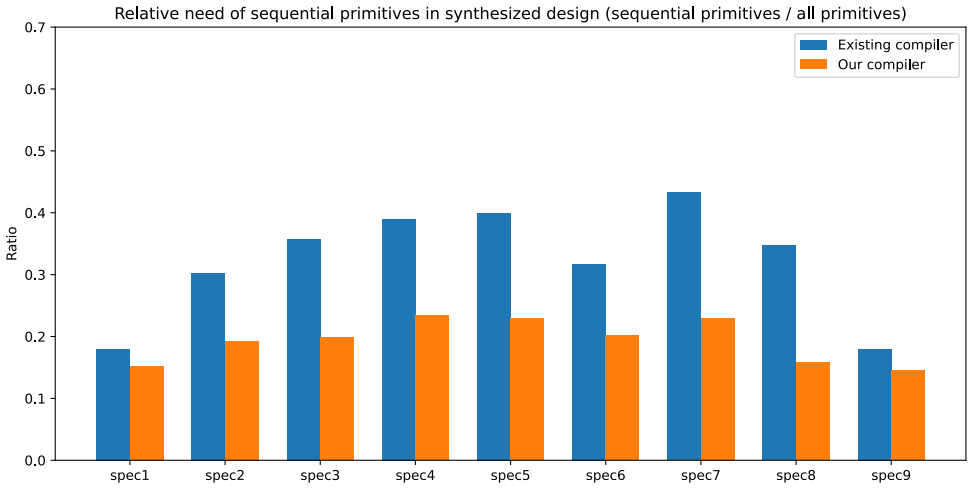


Figure 48: Ratio of sequential primitives over all primitives

Figure 48 illustrates the fraction of total synthesized primitives that are sequential elements, such as flip-flops. The results clearly show that our compiler produces significantly fewer sequential primitives compared to the existing compiler. While the total number of primitives remains relatively similar, our design leans more heavily toward combinational logic.

Interestingly, despite the expectation that pipelining would increase the need for registers (see Section 5.3), our design still uses fewer sequential elements. This suggests that the existing compiler relies more heavily on registers in general.

Whether this is beneficial depends on the target hardware and constraints. For example, on FPGAs, the trade-off between combinational and sequential depends on whether the device is more constrained by registers or by LUTs. Additionally, one design may be better than another in terms of power consumption, maximum frequency, or latency, depending on how sequential and combinational elements are used within the specific design context.

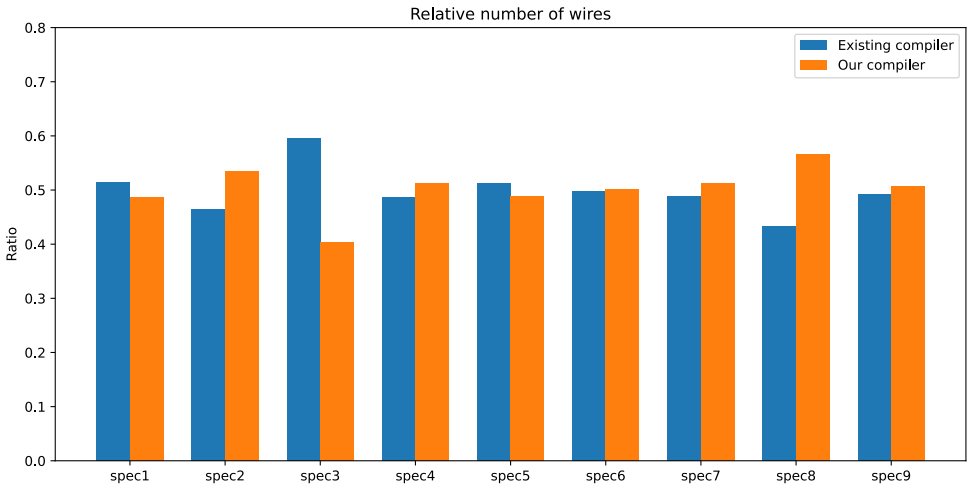


Figure 49: Relative number of synthesized wires

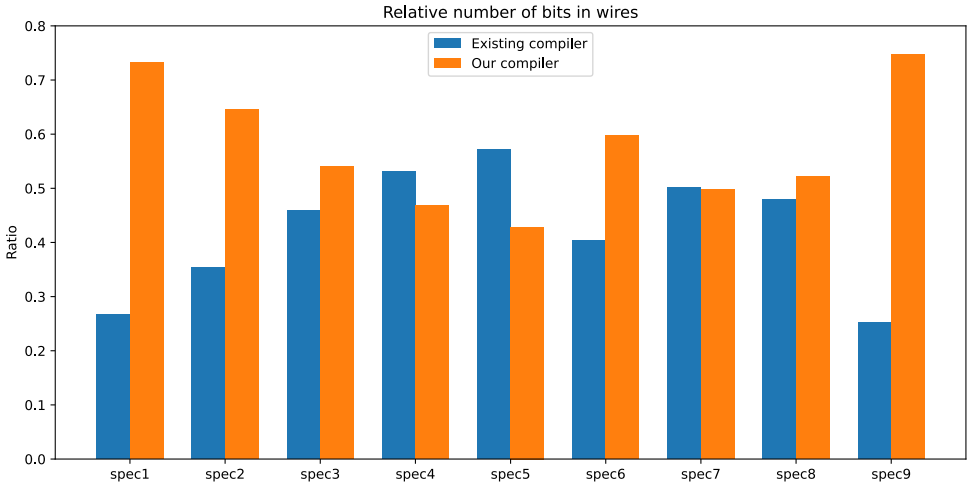


Figure 50: Relative number of total bits in synthesized wires

Figure 49 and Figure 50 show that although the number of synthesized wires is similar between the two designs, our design uses significantly more wire bits. This could be from the specifics in our implementation or from how Clash generates VHDL. Further

investigation is needed to determine the exact cause, and it presents a potential opportunity for optimization.

7.0.5 Summary

We found that pipelining in our architecture leads to a significant improvement in evaluation throughput. Despite using pipeline evaluation, our Clash-based implementation tends to favor combinational circuit primitives over sequential ones, producing significantly less sequential primitives compared to the existing compiler. Additionally, the number of wire bits appears unusually high, indicating a potential area for further investigation.

Chapter 8

Conclusion & Future Work

In this work, we introduced a new hardware architecture that enables pipelined evaluation of streams for RTLola, along with a prototype compiler implemented in Clash. Our evaluation demonstrated a significant increase in throughput due to pipelining. Combined with RTLola's pacing type system enforced in Clash and the ability to target multiple HDL backends, our approach improves the safety, performance, and portability of RTLola hardware monitors.

This work opens several research directions. From a hardware architecture perspective, the current design supports only the core features of RTLola and lacks support for advanced constructs such as stream lifecycles and parameterized streams. The heuristic algorithm used for determining total evaluation order could also be refined to yield further optimizations. On the safety front, while pacing semantics are enforced at the HDL level, incorporating a broader set of RTLola semantics would enhance correctness guarantees. It is also worthwhile exploring the interpreter-style hardware monitors in this direction.

The use of Clash further enables access to Haskell's rich ecosystem for formal verification and property-based testing. Clash also includes verification primitives supporting PSL and SystemVerilog assertions. It could be worthwhile to consider this direction from correctness perspective. As Clash provides a fine-grained control over the generated HDL, the traceability aspect of original specification in the hardware monitor could also be explored with it.

Finally, the current compiler prototype lacks support for several RTLola features, such as mathematical functions, comprehensive aggregation library, etc. Future work could extend the implementation to support broader range of RTLola features. This could be coupled with a real-world case study for a more comprehensive evaluation of the generated monitor.

Chapter A

Appendix

A-A Example RTLola specification with full generated Clash code

A-AA RTLola specification

```
input x : Int
output a := x.offset(by: -1).defaults(to: 0)
            + b.offset(by: -1).defaults(to: 0)
output b := a + c.hold(or: 0)
            + a.offset(by: -2).defaults(to: a + x)
output c @1kHz := b.hold(or: 0) + b.aggregate(over: 0.1s, using: sum)
output d @2kHz := (c.aggregate(over: 0.05s, using: count) < 10)
            and (c.hold(or: 0) > 15)
```

A-AB Generated Clash code

```
{-# LANGUAGE DuplicateRecordFields #-}
{-# LANGUAGE OverloadedRecordDot #-}

module Spec where

import Clash.Prelude

------------------------------------------------------------------

-- input x : Int
-- output a := x.offset(by: -1).defaults(to: 0)
--             + b.offset(by: -1).defaults(to: 0)
-- output b := a + c.hold(or: 0)
--             + a.offset(by: -2).defaults(to: a + x)
-- output c @1kHz := b.hold(or: 0) + b.aggregate(over: 0.1s, using: sum)
-- output d @2kHz := (c.aggregate(over: 0.05s, using: count) < 10)
--             and (c.hold(or: 0) > 15)

------------------------------------------------------------------

-- Evaluation Order
--------------------
-- x, a
```

```
-- b
-- sw(b,c)
-- c
-- sw(c,d)
-- d

-- Memory Window
-----------------
-- window x = 1
-- window a = 3
-- window sw(c,d) = 1
-- window sw(b,c) = 1
-- window c = 1
-- window b = 2
-- window d = 1

-- Pipeline Visualization
-------------------------

-- x,a     |         |         | x,a     |         |         | x,a
-- ------------------------------------------------------------------
--         | b       |         |         | b       |         |
-- ------------------------------------------------------------------
--         |         | sw(b,c) |         |         | sw(b,c) |
-- ------------------------------------------------------------------
--         |         |         | c       |         |         | c
-- ------------------------------------------------------------------
--         |         |         |         | sw(c,d) |         |
-- ------------------------------------------------------------------
--         |         |         |         |         | d       |
-- ------------------------------------------------------------------

-- Nicknames
-------------
-- input0 = x
-- output0 = a
-- output1 = b
-- output2 = c
-- output3 = d
-- sw0 = sw(b,c)
-- sw1 = sw(c,d)

-----------------------------------------------------------------

data ValidInt = ValidInt {
    value :: Int,
    valid :: Bool
} deriving (Generic, NFDataX)

data ValidBool = ValidBool {
    value :: Bool,
    valid :: Bool
```

```
} deriving (Generic, NFDataX)


-- using newtype to avoid flattening of data
-- https://clash-lang.discourse.group/t/how-to-avoid-flattening-of-
fields-in-record/79/5
newtype Inputs = Inputs {
    input0 :: ValidInt
} deriving (Generic, NFDataX)

data Outputs = Outputs {
    output0 :: ValidInt,
    output1 :: ValidInt,
    output2 :: ValidInt,
    output3 :: ValidBool
} deriving (Generic, NFDataX)


class Pacing a where getPacing :: a -> Bool

data PacingIn0 = PacingIn0 Bool deriving (Generic, NFDataX)
data PacingOut0 = PacingOut0 PacingIn0 deriving (Generic, NFDataX)
data PacingOut1 = PacingOut1 PacingIn0 deriving (Generic, NFDataX)
data PacingOut2 = PacingOut2 Bool deriving (Generic, NFDataX)
data PacingOut3 = PacingOut3 Bool deriving (Generic, NFDataX)

instance Pacing PacingIn0 where getPacing (PacingIn0 x) = x
instance Pacing PacingOut0 where getPacing (PacingOut0 x) = getPacing x
instance Pacing PacingOut1 where getPacing (PacingOut1 x) = getPacing x
instance Pacing PacingOut2 where getPacing (PacingOut2 x) = x
instance Pacing PacingOut3 where getPacing (PacingOut3 x) = x

data Pacings = Pacings {
    pacingIn0 :: PacingIn0,
    pacingOut0 :: PacingOut0,
    pacingOut1 :: PacingOut1,
    pacingOut2 :: PacingOut2,
    pacingOut3 :: PacingOut3
} deriving (Generic, NFDataX)

data Slides = Slides {
    slide0 :: Bool,
    slide1 :: Bool
} deriving (Generic, NFDataX)

type Tag = Unsigned 8

data Tags = Tags {
    input0 :: Tag,
    output0 :: Tag,
    output1 :: Tag,
    output2 :: Tag,
```

```
    output3 :: Tag,
    slide0 :: Tag,
    slide1 :: Tag
} deriving (Generic, NFDataX)

type Event = (Inputs, Slides, Pacings)

nullEvent :: Event
nullEvent = (nullInputs, nullSlides, nullPacings)
nullInputs = Inputs (ValidInt 0 False)
nullSlides = Slides False False
nullPacings = Pacings nullPacingIn0
                      nullPacingOut0
                      nullPacingOut1
                      nullPacingOut2
                      nullPacingOut3
nullPacingIn0 = PacingIn0 False
nullPacingOut0 = PacingOut0 nullPacingIn0
nullPacingOut1 = PacingOut1 nullPacingIn0
nullPacingOut2 = PacingOut2 False
nullPacingOut3 = PacingOut3 False


------------------------------------------------------------

type QMemSize = 4

type QData = Event
type QMem = Vec QMemSize QData
type QCursor = Int
type QPush = Bool
type QPop = Bool
type QPushValid = Bool
type QPopValid = Bool

type QState = (QMem, QCursor)
type QInput = (QPush, QPop, QData)
type QOutput = (QPushValid, QPopValid, QData)

queue :: HiddenClockResetEnable dom
    => Signal dom QInput
    -> Signal dom QOutput
queue input = output
    where
        output = bundle (pushValid, popValid, outData)
        state = bundle (buffer, cursor)
        buffer = register (repeat nullEvent :: QMem) nextBufferSignal
        cursor = register 0 nextCursorSignal
        pushValid = register False nextPushValidSignal
        popValid = register False nextPopValidSignal
        outData = register nullEvent nextOutDataSignal
```

```
        nextBufferSignal = nextBuffer
                         <$> buffer
                         <*> bundle (input, cursor)
        nextCursorSignal = nextCursor
                         <$> cursor
                         <*> bundle (input, buffer)
        nextOutDataSignal = nextOutData
                         <$> bundle (input, cursor, buffer)
        nextPushValidSignal = nextPushValid
                         <$> bundle (input, cursor, buffer)
        nextPopValidSignal = nextPopValid <$> bundle (input, cursor)

        nextBuffer :: QMem -> (QInput, QCursor) -> QMem
        nextBuffer buf ((push, pop, qData), cur) = out
            where
                out = case (push, pop) of
                    (True, True) -> qData +>> buf
                    (True, False) -> if cur == length buf
                                     then buf else qData +>> buf
                    (False, _) -> buf

        nextCursor :: QCursor -> (QInput, QMem) -> QCursor
        nextCursor cur ((push, pop, _), buf) = out
            where
                out = case (push, pop) of
                    (True, False) -> if cur == length buf
                                     then cur else cur + 1
                    (False, True) -> if cur == 0 then 0 else cur - 1
                    (_, _) -> cur

        nextOutData :: (QInput, QCursor, QMem) -> QData
        nextOutData ((push, pop, qData), cur, buf) = out
            where
                out = case (push, pop) of
                    (True, True) -> if cur == 0
                                    then qData else buf !! (cur - 1)
                    (False, True) -> if cur == 0
                                     then nullEvent else buf !! (cur - 1)
                    (_, _) -> nullEvent

        nextPushValid :: (QInput, QCursor, QMem) -> QPush
        nextPushValid ((push, pop, _), cur, buf) = out
            where
                out = case (push, pop) of
                    (True, True) -> True
                    (True, False) -> cur /= length buf
                    (False, _) -> False

        nextPopValid :: (QInput, QCursor) -> QPop
        nextPopValid ((push, pop, _), cur) = out
            where
                out = case (push, pop) of
```

```
                    (True, True) -> True
                    (False, True) -> cur /= 0
                    (_, False) -> False


------------------------------------------------------------

-- Clock domain with 2 microseconds period (500 kHz)
-- It has been arbitrarily chosen for both monitor
--  and the verilog testbench simulation
createDomain vSystem{vName="TestDomain", vPeriod=2000}
-- period in nanoseconds

systemClockPeriodNs :: Int
systemClockPeriodNs = fromInteger
    (snatToInteger $ clockPeriod @TestDomain)


hlc :: HiddenClockResetEnable dom
    => Signal dom Inputs
    -> Signal dom (Bool, Event)
hlc inputs = out
    where
        out = bundle (newEvent, event)
        newEvent = hasInput0 .||. timer0Over .||. timer1Over

        event = bundle (inputs, slides, pacings)

        slides = Slides <$> s0
                    <*> s1
        pacings = Pacings <$> pIn0
                    <*> pOut0
                    <*> pOut1
                    <*> pOut2
                    <*> pOut3

        hasInput0 = ((.valid). (.input0)) <$> inputs

        pIn0 = PacingIn0 <$> hasInput0
        pOut0 = PacingOut0 <$> pIn0
        pOut1 = PacingOut1 <$> pIn0
        pOut2 = PacingOut2 <$> timer1Over
        pOut3 = PacingOut3 <$> timer0Over

        s0 = timer1Over
        s1 = timer0Over

        timer0Over = timer0 .>=. period0InNs
        timer0 = timer timer0Over
        period0InNs = 500000
        timer1Over = timer1 .>=. period1InNs
        timer1 = timer timer1Over
```

```
        period1InNs = 1000000

        timer :: HiddenClockResetEnable dom
            => Signal dom Bool
            -> Signal dom Int
        timer reset = register 0 (mux reset (pure deltaTime) nextTime)
            where
                nextTime = timer reset + pure deltaTime
                deltaTime = systemClockPeriodNs


------------------------------------------------------------------

-- To avoid duplidate tags in a window
-- maxTag must be at least the size of the maximum window
-- Also to avoid having to do modulo operations
-- maxTag must be at least as big as the largest offset
maxTag = 102 :: Tag
invalidTag = maxTag + 1

getOffset :: KnownNat n => Vec n (Tag, a) -> Tag -> Tag -> a -> a
getOffset win tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = case findIndex (\(t, _) -> t == offsetTag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getMatchingTag :: KnownNat n => Vec n (Tag, a) -> Tag -> a -> a
getMatchingTag win tag dflt = out
    where
        out = case findIndex (\(t, _) -> t == tag) win of
            Just i -> let (_, v) = win !! i in v
            Nothing -> dflt

getOffsetFromNonVec :: (Tag, a) -> Tag -> Tag -> a -> a
getOffsetFromNonVec (winTag, winData) tag offset dflt = out
    where
        offsetTag = earlierTag tag offset
        out = if offsetTag == winTag then winData else dflt

getMatchingTagFromNonVec :: (Tag, a) -> Tag -> a -> a
getMatchingTagFromNonVec (tag, dta) tagToMatch dflt =
    if tag == tagToMatch then dta else dflt

getLatestValue :: KnownNat n => Vec (n + 1) (Tag, a) -> a -> a
getLatestValue win dflt =
    let (tag, dta) = last win
    in if tag == invalidTag then dflt else dta

getLatestValueFromNonVec :: (Tag, a) -> a -> a
getLatestValueFromNonVec (tag, dta) dflt =
```

```
    if tag == invalidTag then dflt else dta

earlierTag :: Tag -> Tag -> Tag
earlierTag curTag cyclesBefore =
    if curTag > cyclesBefore
    then curTag - cyclesBefore
    else curTag - cyclesBefore + maxTag

delayFor :: forall dom n a .
    (HiddenClockResetEnable dom, KnownNat n, NFDataX a)
    => SNat n
    -> a
    -> Signal dom a
    -> Signal dom a
delayFor n initVal sig = last delayedVec
    where
        delayedVec :: Vec (n + 1) (Signal dom a)
        delayedVec = iterateI (delay initVal) sig


llc :: HiddenClockResetEnable dom
    => Signal dom (Bool, Event)
    -> Signal dom (Bool, Outputs)
llc event = bundle (toPop, outputs)
    where
        (isValidEvent, poppedEvent) = unbundle event

        isPipelineReady = pipelineReady startNewPipeline
        startNewPipeline = mux (isPipelineReady .&&. isValidEvent)
                               (pure True) (pure False)
        toPop = isPipelineReady .&&. not <$> startNewPipeline

        (inputs, slides, pacings) = unbundle poppedEvent

        input0 = (.input0) <$> inputs

        slide0 = (.slide0) <$> slides
        slide1 = (.slide1) <$> slides

        pIn0 = (.pacingIn0) <$> pacings
        pOut0 = (.pacingOut0) <$> pacings
        pOut1 = (.pacingOut1) <$> pacings
        pOut2 = (.pacingOut2) <$> pacings
        pOut3 = (.pacingOut3) <$> pacings

        tIn0 = genTag (getPacing <$> pIn0)
        tOut0 = genTag (getPacing <$> pOut0)
        tOut1 = genTag (getPacing <$> pOut1)
        tSw0 = genTag (getPacing <$> pOut1)
        tOut2 = genTag (getPacing <$> pOut2)
        tSw1 = genTag (getPacing <$> pOut2)
        tOut3 = genTag (getPacing <$> pOut3)
```

```
        -- tag generation takes 1 cycle so we need to delay the input
        input0Data = delay 0 (((.value). (.input0)) <$> inputs)

        -- delayed tags to be used in different levels
        tagsDefault = Tags
                nullT
                nullT
                nullT
                nullT
                nullT
                nullT
                nullT
        curTags = Tags
                <$> tIn0
                <*> tOut0
                <*> tOut1
                <*> tOut2
                <*> tOut3
                <*> tSw0
                <*> tSw1
        curTagsLevel1 = delayFor d1 tagsDefault curTags
        curTagsLevel2 = delayFor d2 tagsDefault curTags
        curTagsLevel3 = delayFor d3 tagsDefault curTags
        curTagsLevel4 = delayFor d4 tagsDefault curTags
        curTagsLevel5 = delayFor d5 tagsDefault curTags
        curTagsLevel6 = delayFor d6 tagsDefault curTags
        nullT = invalidTag

        enIn0 = delayFor d1 nullPacingIn0 pIn0
        enOut0 = delayFor d1 nullPacingOut0 pOut0
        enOut1 = delayFor d2 nullPacingOut1 pOut1
        enSw0 = delayFor d3 nullPacingOut1 pOut1
        sld0 = delayFor d3 False slide0
        enOut2 = delayFor d4 nullPacingOut2 pOut2
        enSw1 = delayFor d5 nullPacingOut2 pOut2
        sld1 = delayFor d5 False slide1
        enOut3 = delayFor d6 nullPacingOut3 pOut3

        output0Aktv = delayFor d7 False (getPacing <$> pOut0)
        output1Aktv = delayFor d7 False (getPacing <$> pOut1)
        output2Aktv = delayFor d7 False (getPacing <$> pOut2)
        output3Aktv = delayFor d7 False (getPacing <$> pOut3)

        -- Evaluation of input windows: level 0
        input0Win = input0Window enIn0 tIn0 input0Data

        -- Evaluation of output 0: level 0
        out0 = outputStream0 enOut0
            tOut0
            out0Data0
            out0Data1
```

```
        out0Data0 = getOffsetFromNonVec
            <$> input0Win
            <*> tIn0
            <*> (pure 1)
            <*> out0Data0Dflt
        out0Data0Dflt = pure (0)
        out0Data1 = getOffset
            <$> out1
            <*> tOut1
            <*> (pure 1)
            <*> out0Data1Dflt
        out0Data1Dflt = pure (0)

        -- Evaluation of output 1: level 1
        out1 = outputStream1 enOut1
            ((.output1) <$> curTagsLevel1)
            out1Data0
            out1Data1
            out1Data2
        out1Data0 = getMatchingTag
            <$> out0
            <*> ((.output0) <$> curTagsLevel1)
            <*> (pure (0))
        out1Data1 = getLatestValueFromNonVec
            <$> out2
            <*> out1Data1Dflt
        out1Data1Dflt = pure (0)
        out1Data2 = getOffset
            <$> out0
            <*> ((.output0) <$> curTagsLevel1)
            <*> (pure 2)
            <*> out1Data2Dflt
        out1Data2Dflt = out1Data2DfltData0 + out1Data2DfltData1
        out1Data2DfltData0 =  getMatchingTag
            <$> out0
            <*> ((.output0) <$> curTagsLevel1)
            <*> (pure (0))
        (_, out1Data2DfltData1) = unbundle input0Win

        -- Evaluation of output 2: level 3
        out2 = outputStream2 enOut2
            ((.output2) <$> curTagsLevel3)
            out2Data0
            out2Data1
        out2Data0 = getLatestValue
            <$> out1
            <*> out2Data0Dflt
        out2Data0Dflt = pure (0)
        (_, out2Data1) = unbundle sw0

        -- Evaluation of output 3: level 5
        out3 = outputStream3 enOut3
```

```
        ((.output3) <$> curTagsLevel5)
        out3Data0
        out3Data1
    (_, out3Data0) = unbundle sw1
    out3Data1 = getLatestValueFromNonVec
        <$> out2
        <*> out3Data1Dflt
    out3Data1Dflt = pure (0)

    -- Evaluation of sliding window 0: level 2
    sw0 = slidingWindow0 enSw0 sld0
        ((.slide0) <$> curTagsLevel2) sw0Data
    sw0Data = getMatchingTag
        <$> out1
        <*> ((.output1) <$> curTagsLevel2)
        <*> (pure 0)

    -- Evaluation of sliding window 1: level 4
    sw1 = slidingWindow1 enSw1 sld1
        ((.slide1) <$> curTagsLevel4) sw1Data
    (_, sw1Data) = unbundle out2

    -- Outputing all results: level 6
    output0 = ValidInt <$> output0Data <*> output0Aktv
    output0Data = getMatchingTag
        <$> out0
        <*> ((.output0)
        <$> curTagsLevel6)
        <*> (pure 0)
    output1 = ValidInt <$> output1Data <*> output1Aktv
    output1Data = getMatchingTag
        <$> out1
        <*> ((.output1)
        <$> curTagsLevel6)
        <*> (pure 0)
    output2 = ValidInt <$> output2Data <*> output2Aktv
    (_, output2Data) = unbundle out2
    output3 = ValidBool <$> output3Data <*> output3Aktv
    (_, output3Data) = unbundle out3

    outputs = Outputs
        <$> output0
        <*> output1
        <*> output2
        <*> output3


    genTag :: HiddenClockResetEnable dom
        => Signal dom Bool
        -> Signal dom Tag
    genTag en = t
        where
```

```
                t = register 1 (mux en next_t t)
                next_t = mux (t .==. (pure maxTag)) (pure 1) (t + 1)


pipelineReady :: HiddenClockResetEnable dom
    => Signal dom Bool
    -> Signal dom Bool
pipelineReady rst = toWait .==. pure 0
    where
        waitTime = pure 2 :: Signal dom Int
        toWait = register (0 :: Int) next
        next = mux rst waitTime
                (mux (toWait .>. pure 0) (toWait - 1) toWait)



input0Window :: HiddenClockResetEnable dom
    => Signal dom PacingIn0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, Int)
input0Window en tag val = result
    where result = register (invalidTag, 0)
                    (mux (getPacing <$> en) (bundle (tag, val)) result)



outputStream0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut0
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom (Vec 3 (Tag, Int))
outputStream0 en tag in0_0 out1_1 = result
    where
        result = register (repeat (invalidTag, 0))
                (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (in0_0 + out1_1)


outputStream1 :: HiddenClockResetEnable dom
    => Signal dom PacingOut1
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom Int
    -> Signal dom (Vec 2 (Tag, Int))
outputStream1 en tag out0_00 out2_01 out0_1 = result
    where
        result = register (repeat (invalidTag, 0))
```

```
                (mux (getPacing <$> en) next result)
        next = (<<+) <$> result <*> nextValWithTag
        nextValWithTag = bundle (tag, nextVal)
        nextVal = ((out0_00 + out2_01) + out0_1)


outputStream2 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Vec 101 Int)
    -> Signal dom (Tag, Int)
outputStream2 en tag out1_0 sw0 = result
    where
        result = register (invalidTag, 0)
                (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (out1_0 + (merge0 <$> sw0))
        merge0 :: Vec 101 Int -> Int
        merge0 win = fold windowFunc0 (tail win)


outputStream3 :: HiddenClockResetEnable dom
    => Signal dom PacingOut3
    -> Signal dom Tag
    -> Signal dom (Vec 101 Int)
    -> Signal dom Int
    -> Signal dom (Tag, Bool)
outputStream3 en tag sw1 out2_10 = result
    where
        result = register (invalidTag, False)
                (mux (getPacing <$> en) nextValWithTag result)
        nextValWithTag = bundle (tag, nextVal)
        nextVal = (((merge1 <$> sw1) .<. (10)) .&&. (out2_10 .>. (15)))
        merge1 :: Vec 101 Int -> Int
        merge1 win = fold windowFunc0 (tail win)



windowFunc0 :: Int -> Int -> Int
windowFunc0 acc item = acc + item

windowFunc1 :: Int -> Int -> Int
windowFunc1 acc item = acc + 1


slidingWindow0 :: HiddenClockResetEnable dom
    => Signal dom PacingOut1
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 101 Int))
```

```
slidingWindow0 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 101 Int

        nextWindow :: Vec 101 Int
            -> Bool
            -> Bool
            -> Int
            -> Vec 101 Int
        nextWindow win toSlide newData dta = out
            where
                out = case (toSlide, newData) of
                    (False, False) -> win
                    (False, True) -> lastBucketUpdated
                    (True, False) -> 0 +>> win
                    (True, True) -> 0 +>> lastBucketUpdated
                lastBucketUpdated =
                    replace 0 (windowFunc0 (head win) dta) win

slidingWindow1 :: HiddenClockResetEnable dom
    => Signal dom PacingOut2
    -> Signal dom Bool
    -> Signal dom Tag
    -> Signal dom Int
    -> Signal dom (Tag, (Vec 101 Int))
slidingWindow1 newData slide tag inpt = window
    where
        window = register (invalidTag, dflt) (mux en next window)
        next = bundle (tag, nextWindow <$> (snd <$> window)
                <*> slide <*> (getPacing <$> newData) <*> inpt)
        en = (getPacing <$> newData) .||. slide
        dflt = repeat 0 :: Vec 101 Int

        nextWindow :: Vec 101 Int
            -> Bool
            -> Bool
            -> Int
            -> Vec 101 Int
        nextWindow win toSlide newData dta = out
            where
                out = case (toSlide, newData) of
                    (False, False) -> win
                    (False, True) -> lastBucketUpdated
                    (True, False) -> 0 +>> win
                    (True, True) -> 0 +>> lastBucketUpdated
                lastBucketUpdated =
                    replace 0 (windowFunc1 (head win) dta) win
```

```
------------------------------------------------------------

monitor :: HiddenClockResetEnable dom
    => Signal dom Inputs
    -> Signal dom Outputs
monitor inputs = outputs
    where
        (newEvent, event) = unbundle (hlc inputs)

        (qPushValid, qPopValid, qPopData) =
            unbundle (queue (bundle (qPush, qPop, qInptData)))
        qPush = newEvent
        qPop = toPop
        qInptData = event

        (toPop, outputs) = unbundle (llc (bundle (qPopValid, qPopData)))


------------------------------------------------------------

topEntity :: Clock TestDomain
    -> Reset TestDomain
    -> Enable TestDomain
    -> Signal TestDomain Inputs
    -> Signal TestDomain Outputs
topEntity clk rst en inputs =
    exposeClockResetEnable (monitor inputs) clk rst en
```

A-B RTLola Specifications used in evaluation

A-BA Spec1: Drone example

This specification has been adapted from the RTLola tutorial [37] in the RTLola playground to fit the currenlty supported features in this compiler and the existing compiler.

```
input acceleration_x: Int64
input gps_sats: UInt64
input lat_gps: Int64

output acceleration_x_periodic @2kHz := acceleration_x.hold(or: 0)

output acceleration_x_rising :=
    delta(acceleration_x_periodic, dft: acceleration_x_periodic) > 5

output acceleration_x_sinking :=
    delta(acceleration_x_periodic, dft: acceleration_x_periodic) < -5

output acceleration_x_direction_change :=
```

```
    (acceleration_x_rising
      ∧ acceleration_x_sinking.offset(by: −1).defaults(to: false))
    ∨ ( acceleration_x_sinking
      ∧ acceleration_x_rising.offset(by: −1).defaults(to: false))

output acceleration_x_changes @2kHz :=
    acceleration_x_direction_change.aggregate(over: 0.05s, using: count)

output trigger_acc := acceleration_x_changes > 5

output gps_missed_beat: Bool @ 2kHz :=
    lat_gps.aggregate(over: 0.055s, using: count) < 1

output gps_medium_loss: Bool @ 0.1kHz :=
    lat_gps.aggregate(over: 0.01s, using: count) < 15
    && lat_gps.aggregate(over: 0.01s, using: count) >= 10

output gps_high_loss: Bool @ 0.1kHz :=
    lat_gps.aggregate(over: 0.01s, using: count) < 10
    && lat_gps.aggregate(over: 0.01s, using: count) >= 5

output gps_very_high_loss: Bool @ 0.1kHz :=
    lat_gps.aggregate(over: 0.01s, using: count) < 5

output trigger_gps_sats: Bool @ 0.1kHz := gps_sats.hold(or: 0)  < 6
```

A-BB Spec2: Intruder

This specification has been adapted from the RTLola tutorial [37] in the RTLola playground to fit the currenlty supported features in this compiler and the existing compiler.

```
input lat: Int
input lon: Int
constant intruder_lat: Int := 249
constant intruder_lon: Int := 23

output distance: Int := (intruder_lat − lat) + (intruder_lon − lon)
output closer: Bool := distance.offset(by: −1).defaults(to: distance) >=
distance
output trigger_closer := closer && distance < 1
output is_good @1kHz :=
    trigger_closer.aggregate(over: 0.01s, using: count) < 5
```

A-BC Spec3: Sensor data validation

This specification has been adapted from [20] to fit the newer syntax of RTLola and the currenlty supported features in this compiler and the existing compiler.

```
input gps_x : Int
input num_satellites: Int
input imu_acc_x: Int
```

```
output gps_emitted_enough @1Hz := gps_x.aggregate(over: 3s, using: count)
< 10
output few_satellites := num_satellites < 9
output is_unreliable_gps_data @1Hz :=
    few_satellites.aggregate(over: 5s, using: count) > 12
```

A-BD Spec4: Cycle from hold access

```
input x: Int
input y: Int

output a := x + 1
output b @1kHz := y.hold(or: 1) + d.hold(or: 2)
output c @2kHz := b.hold(or: 10) + a.hold(or: 11)
output d @(x & y) := c.hold(or: 0) + 1

output e @1kHz := e.offset(by: -1).defaults(to: 0) + 1
output f := e + 1
output g := f + 1
```

A-BE Spec5: Simple case with pipeline_wait

```
input x: Int
output a := x + c.offset(by: -1).defaults(to: 0)
output b := a + 1
output c := b + 1
output xx @1kHz := a.hold(or: 0)
```

A-BF Spec6: Recursive defaults

```
input x : Int
input y : Int

output a := y + x.offset(by: -2).defaults(to:
                    y.offset(by: -1).defaults(to: 10)
                    + a.offset(by: -1).defaults(to:
                        x.offset(by: -1).defaults(to: 20)))
output counts @1kHz := a.aggregate(over: 0.01s, using: count)
```

A-BG Spec7: Only forward offsets

```
input x : Int

output a := x.offset(by: -1).defaults(to: 0) + 1
output b := a.offset(by: -3).defaults(to: 0)
            + x.offset(by: -2).defaults(to: 0)
output c := a.offset(by: -1).defaults(to:
                b.offset(by: -1).defaults(to: 0))
output d @1kHz := c.hold(or: 0)
```

A-BH Spec8: Offset with loop

```
input x : Int

output a := x.offset(by: -1).defaults(to: 0)
               + b.offset(by: -1).defaults(to: 0)
               + c.offset(by: -1).defaults(to: 0)
output b := a.offset(by: -3).defaults(to: 0)
            + x.offset(by: -2).defaults(to: 0)
output c := x + a + b
output d @1kHz := c.hold(or: 0)
```

A-BI Spec9: All major supported

```
input x : Int
output a := x.offset(by: -1).defaults(to: 0)
            + b.offset(by: -1).defaults(to: 0)
output b := a + c.hold(or: 0)
            + a.offset(by: -2).defaults(to: a + x)
output c @1kHz := b.hold(or: 0) + b.aggregate(over: 0.1s, using: sum)
output d @2kHz := (c.aggregate(over: 0.05s, using: count) < 10)
            and (c.hold(or: 0) > 15)
```

A-C Finding root nodes where evaluation starts

Roots are the nodes from where the evaluation should start. Normally they would be the input nodes. However, there can also be roots that are output nodes.

Example A-C.1 (Non-input root node)

```
output a @1Hz := a.offset(by: -1).defaults(to: 0) + 1
output b := a + 1
output c := b + 1
```

Here all the nodes are periodic with the evaluation starting from `a`. So the evaluation order must be $a \to b \to c$, with `a` as the root node.

Algorithm A-C.1 (Find roots)

```
FindRoots(mir):
roots ⟵ set of all nodes
for node in nodes do
    if node ∈ roots then
        reachable ⟵ ReachableNodes({node}, mir, node_cond)
        roots ⟵ roots
        reachable ∪ {node}
connected_roots ⟵ empty list
marked ⟵ array of false values of length |roots|
for i from 0 to |roots| - 1 do
    if not marked[i] then
        group ⟵ [roots[i]]
        for j from i+1 to |roots| - 1 do
            if not marked[j] and AreConnected(roots[i], roots[j], mir, node_cond)
            then
                append roots[j] to group
                marked[j] ⟵ true
            append group to connected_roots
return connected_roots
```

Bibliography

[1] B. d'Angelo *et al.*, "LOLA: runtime monitoring of synchronous systems," in *12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, 2005, pp. 166–174.

[2] P. Faymonville, B. Finkbeiner, S. Schirmer, and H. Torfah, "A stream-based specification language for network monitoring," in *International Conference on Runtime Verification*, 2016, pp. 152–168.

[3] P. Faymonville, B. Finkbeiner, M. Schwenger, and H. Torfah, "Real-time stream-based monitoring," *arXiv preprint arXiv:1711.03829*, 2017.

[4] L. Convent, S. Hungerecker, M. Leucker, T. Scheffel, M. Schmitz, and D. Thoma, "TeSSLa: temporal stream-based specification language," in *Formal Methods: Foundations and Applications: 21st Brazilian Symposium, SBMF 2018, Salvador, Brazil, November 26–30, 2018, Proceedings 21*, 2018, pp. 144–162.

[5] F. Gorostiaga and C. Sánchez, "Striver: Stream runtime verification for real-time event-streams," in *International Conference on Runtime Verification*, 2018, pp. 282–298.

[6] J. Baumeister, B. Finkbeiner, M. Schwenger, and H. Torfah, "FPGA stream-monitoring of real-time properties," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–24, 2019.

[7] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, 1977, pp. 46–57. doi: 10.1109/SFCS.1977.32.

[8] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Workshop on logic of programs*, 1981, pp. 52–71.

[9] K. Havelund and G. Roşu, "Synthesizing monitors for safety properties," in *Tools and Algorithms for the Construction and Analysis of Systems: 8th International Conference, TACAS 2002 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8–12, 2002 Proceedings 8*, 2002, pp. 342–356.

[10] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 20, no. 4, pp. 1–64, 2011.

[11] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-time systems*, vol. 2, no. 4, pp. 255–299, 1990.

[12] O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *International symposium on formal techniques in real-time and fault-tolerant systems*, 2004, pp. 152–166.

[13] T. Gautier, P. Le Guernic, and L. Besnard, *Signal: A declarative language for synchronous programming of real-time systems*. Springer, 1987.

[14] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, 1991, doi: 10.1109/5.97300.

[15] G. Berry, "The foundations of Esterel," 2000.

[16] L. Pike, A. Goodloe, R. Morisset, and S. Niller, "Copilot: A hard real-time runtime monitor," in *International Conference on Runtime Verification*, 2010, pp. 345–359.

[17] J. Schumann, P. Moosbrugger, and K. Y. Rozier, "R2U2: monitoring and diagnosis of security threats for unmanned aerial systems," in *Runtime Verification: 6th International Conference, RV 2015, Vienna, Austria, September 22-25, 2015. Proceedings*, 2015, pp. 233–249.

[18] A. Dutle *et al.*, "From requirements to autonomous flight: an overview of the monitoring ICAROUS project," *arXiv preprint arXiv:2012.03745*, 2020.

[19] I. Perez, A. Mavridou, T. Pressburger, A. Goodloe, and D. Giannakopoulou, "Integrating FRET with Copilot: Automated Translation of Natural Language Requirements to Runtime Monitors," 2022.

[20] J. Baumeister, B. Finkbeiner, S. Schirmer, M. Schwenger, and C. Torens, "RTLola cleared for take-off: monitoring autonomous aircraft," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*, 2020, pp. 28–39.

[21] M. Schwenger, "Monitoring cyber-physical systems: From design to integration," in *International Conference on Runtime Verification*, 2020, pp. 87–106.

[22] S. P. Jones, *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[23] R. Krook, J. Hui, B. J. Svensson, S. A. Edwards, and K. Claessen, "Creating a Language for Writing Real-Time Applications for the Internet of Things," in *2022 20th ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2022, pp. 1–20. doi: 10.1109/MEMOCODE57689.2022.9954383.

[24] F. Dedden, "Compiling an Haskell EDSL to C: A new C back-end for the Copilot runtime verification framework," 2018.

[25] A. Dahan *et al.*, "Combining system level modeling with assertion based verification," in *Sixth international symposium on quality electronic design (isqed'05)*, 2005, pp. 310–315.

[26] H. Lu and A. Forin, "The design and implementation of P2V, an architecture for zero-overhead online verification of software programs," 2007.

[27] P. H. Cheung and A. Forin, "A C-language binding for PSL," in *Embedded Software and Systems: Third International Conference, ICESS 2007, Daegu, Korea, May 14-16, 2007. Proceedings 3*, 2007, pp. 584–591.

[28] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu, "Hardware runtime monitoring for dependable cots-based real-time embedded systems," in *2008 Real-Time Systems Symposium*, 2008, pp. 481–491.

[29] P. O. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, "An overview of the MOP runtime verification framework," *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.

[30] M. Boule and Z. Zilic, "Automata-based assertion-checker synthesis of PSL properties," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 13, no. 1, pp. 1–21, 2008.

[31] B. Finkbeiner and L. Kuhtz, "Monitor circuits for LTL with bounded and unbounded future," in *International Workshop on Runtime Verification*, 2009, pp. 60–75.

[32] S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković, "From signal temporal logic to FPGA monitors," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2015, pp. 218–227.

[33] T. Reinbacher, K. Y. Rozier, and J. Schumann, "Temporal-logic based runtime observer pairs for system health management of real-time systems," in *Tools and Algorithms for the Construction and Analysis of Systems: 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings 20*, 2014, pp. 357–372.

[34] J. Geist, K. Y. Rozier, and J. Schumann, "Runtime observer pairs and Bayesian network reasoners on-board FPGAs: flight-certifiable system health management for embedded systems," in *International Conference on Runtime Verification*, 2014, pp. 215–230.

[35] H. Torfah, "Stream-based monitors for real-time properties," in *Runtime Verification: 19th International Conference, RV 2019, Porto, Portugal, October 8–11, 2019, Proceedings 19*, 2019, pp. 91–110.

[36] J. Baumeister, B. Finkbeiner, M. Kruse, and M. Schwenger, "Automatic optimizations for stream-based monitoring languages," in *International Conference on Runtime Verification*, 2020, pp. 451–461.

[37] J. Baumeister, B. Finkbeiner, F. Kohn, and F. Scheerer, "A Tutorial on Stream-Based Monitoring," in *International Symposium on Formal Methods*, 2024, pp. 624–648.

[38] L. Convent, S. Hungerecker, T. Scheffel, M. Schmitz, D. Thoma, and A. Weiss, "Hardware-based runtime verification with embedded tracing units and stream processing," in *International Conference on Runtime Verification*, 2018, pp. 43–63.

[39] H. Kallwies, M. Leucker, M. Schmitz, A. Schulz, D. Thoma, and A. Weiss, "TeSSLa–an ecosystem for runtime verification," in *International Conference on Runtime Verification*, 2022, pp. 314–324.

[40] M. Schmitz, "Efficient Implementation of Stream Transformations," 2022.

[41] F. Ahishakiye, J. I. R. Jarabo, V. K. I. Pun, and V. Stolz, "Hardware-assisted online data race detection," *Formal Methods in Outer Space: Essays Dedicated to Klaus Havelund on the Occasion of His 65th Birthday*, pp. 108–126, 2021.

[42] P. Schmid, "Runtime verification with tessla on enzian," 2019.

[43] D. Cock *et al.*, "Enzian: an open, general, CPU/FPGA platform for systems software research," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 434–451.

[44] A. Benny, S. Chandran, R. Kalayappan, R. Phawade, and P. P. Kurur, "faRM-LTL: A Domain-Specific Architecture for Flexible and Accelerated Runtime Monitoring of LTL Properties," in *International Conference on Runtime Verification*, 2024, pp. 109–127.

[45] N. Wirth, "Hardware compilation: Translating programs into circuits," *Computer*, vol. 31, no. 6, pp. 25–31, 1998.

[46] "Verilator." Accessed: Oct. 31, 2024. [Online]. Available: https://www.veripool.org/verilator/

[47] "Gtkwave." Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/gtkwave/gtkwave

[48] K. E. Murray, M. A. Elgammal, V. Betz, T. Ansell, K. Rothman, and A. Comodi, "SymbiFlow and VPR: An open-source design flow for commercial and novel FPGAs," *IEEE Micro*, vol. 40, no. 4, pp. 49–57, 2020.

[49] C. Wolf, J. Glaser, and J. Kepler, "Yosys-a free Verilog synthesis suite," in *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[50] S. Williams and M. Baxter, "Icarus verilog: open-source verilog more than a year later," *Linux Journal*, vol. 2002, no. 99, p. 3, 2002.

[51] "Vivado HLS." Accessed: Oct. 31, 2024. [Online]. Available: https://docs.amd.com/v/u/2016.4-English/ug902-vivado-high-level-synthesis

[52] "Intel FPGA SDK for OpenCL." Accessed: Oct. 31, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683846/22-4/fpga-programming-flow.html

[53] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: hardware design in Haskell," *Acm Sigplan Notices*, vol. 34, no. 1, pp. 174–184, 1998.

[54] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE'04.*, 2004, pp. 69–70.

[55] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards, "C? ash: Structural descriptions of synchronous hardware using haskell," in *2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010, pp. 714–721.

[56] C. Baaij, "C\lambdaash: From haskell to hardware", 2009.

[57] M. Kooijman, “Haskell as a higher order structural hardware description language,” 2009.

[58] C. P. R. Baaij, “Digital circuit in C\lambdaaSH: functional specifications and type-directed synthesis”, 2015.

[59] P. Hanrahan, “Github - phanrahan/magma: magma circuits.” Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/phanrahan/magma

[60] J. Decaluwe, “MyHDL: a python-based hardware description language,” *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[61] “Github - amaranth: A modern hardware definition language and toolchain based in python.” Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/amaranth-lang/amaranth

[62] A. Ray, B. Devlin, F. Y. Quah, and R. Yesantharao, “Hardcaml: An OCaml Hardware Domain-Specific Language for Efficient and Robust Design,” *arXiv preprint arXiv:2312.15035*, 2023.

[63] J. Bachrach *et al.*, “Chisel: constructing hardware in a scala embedded language,” in *Proceedings of the 49th Annual Design Automation Conference*, 2012, pp. 1216–1225.

[64] “Github - SpinalHDL: A high level hardware description language.” Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/SpinalHDL

[65] O. Port and Y. Etsion, “DFiant: A dataflow hardware description language,” in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–4. doi: 10.23919/FPL.2017.8056858.

[66] “TL-Verilog.” Accessed: Oct. 31, 2024. [Online]. Available: https://www.redwoodeda.com/tl-verilog

[67] S. Eldridge *et al.*, “MLIR as hardware compiler infrastructure,” in *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[68] “Github - llvm/circt: Circuit IR compilers and tools.” Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/llvm/circt

[69] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, “LLHD: A multi-level intermediate representation for hardware description languages,” in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 258–271.

[70] R. Nigam, S. Thomas, Z. Li, and A. Sampson, “A compiler infrastructure for accelerator generators,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 804–817.

[71] F. Schuiki, “Github - fabianschuiki/moore: A hardware compiler based on LLHD and CIRCT.” Accessed: Oct. 31, 2024. [Online]. Available: https://github.com/fabianschuiki/moore

[72] M. Ceresa, F. Gorostiaga, and C. Sánchez, "Declarative stream runtime verification (hLola)," in *Asian Symposium on Programming Languages and Systems*, 2020, pp. 25–43.

[73] M. Schwenger, "Statically-analyzed stream monitoring for cyber-physical Systems.," 2022.

[74] M. Schwenger, "Let's not trust experience blindly: formal monitoring of humans and other CPS," 2020.

[75] J. Baumeister, "Tracing correctness: a practical approach to traceable runtime monitoring," 2020.

[76] P. Wadler, "The essence of functional programming," in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1992, pp. 1–14.

[77] S. L. Peyton Jones, *The implementation of functional programming languages (prentice-hall international series in computer science)*. Prentice-Hall, Inc., 1987.

[78] S. L. P. Jones, "Compiling Haskell by program transformation: A report from the trenches," in *European Symposium on Programming*, 1996, pp. 18–44.

[79] "Clash Tutorial." Accessed: Oct. 31, 2024. [Online]. Available: https://hackage-content.haskell.org/package/clash-prelude-1.8.2/docs/Clash-Tutorial.html

[80] G. Érdi, "Retrocomputing with Clash." [Online]. Available: https://gergo.erdi.hu/retroclash/