RICE UNIVERSITY

Language Support for Real-time Data Processing

By

Lingkun Kong

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE

Konstantinos Mamouras
Konstantinos Mamouras (Aug 2, 2023 11:01 CDT)

Konstantinos Mamouras (Chair)
Assistant Professor of Computer Science

Robert Cartwright
Robert Cartwright (Aug 7, 2023 12:19 CDT)

Robert Cartwright
Professor of Computer Science

Kaiyuan Yang

Kaiyuan Yang
Associate Professor of Electrical and
Computer Engineering

HOUSTON, TEXAS

August 2023

ABSTRACT

Language Support for Real-time Data Processing

by

Lingkun Kong

Recent technological advances are causing an enormous proliferation of streaming data, i.e., data that is generated in real time. Such data is produced at an overwhelming rate that cannot be effectively processed using traditional techniques. This thesis aims to provide programming language support for real-time data processing through three approaches: (1) creating a language for specifying complex computations over real-time data streams, (2) employing software-hardware co-design to create an efficient accelerator for matching regular patterns in a streaming setting, and (3) designing a programming system for parallel stream processing that guarantees the preservation of sequential semantics.

The first part of this thesis introduces StreamQL, a high-level language for specifying complex streaming computations through the composition of stream transformations. StreamQL integrates relational, dataflow, and temporal constructs, offering an expressive and modular approach for programming streaming computations. Performance comparisons against popular streaming engines show that the StreamQL library consistently achieves higher throughput, making it a useful tool for prototyping complex real-world streaming algorithms.

The second part of this thesis focuses on hardware acceleration for regular pattern matching, specifically targeting the matching of regular expressions with bounded

repetitions. A hardware architecture inspired by nondeterministic counter automata is presented. This architecture uses counter and bit vector modules to efficiently handle bounded repetitions. A regex-to-hardware compiler provides static analysis over regular expressions and uses the results of the analysis to translate them into memory-efficient hardware-recognizable programs. Experimental results show that our solution provides significant improvements in energy efficiency and area reduction compared to existing hardware designs.

Finally, this thesis presents a novel programming system for parallelizing the processing of streaming data on multicore CPUs while preserving the sequential semantics. This system addresses challenges in preserving the sequential semantics when dealing with duplicate timestamps, dynamic item rates, and non-linear task parallelism. A Rust library called ParaStream is developed to support semantics-preserving parallelism in stream processing. ParaStream outperforms state-of-the-art tools in terms of single-threaded throughput and scalability. Real-world benchmarks show substantial performance gains with increasing degrees of parallelism, highlighting the practicality and efficiency of ParaStream.

Acknowledgments

My deepest appreciation goes to my advisor, Konstantinos Mamouras, for his exceptional patience and insightful guidance during my research journey. Over five years, he set himself as an example and guided me to be a better researcher with his vast knowledge and extensive expertise. Through his mentorship, he has helped me to improve my research taste, enabling me to concentrate on the more critical problems.

During my PhD, I had the privilege of receiving guidance from numerous additional mentors. I would like to extend my thanks to Corky Cartwright for expanding my perspective on programming languages and their future design, Kaiyuan Yang for his expertise in hardware circuit design, John Mellor-Crummey for teaching me the subtleties of parallel programming system implementation, Tracy Volz for helping me improve my presentation skills, and Xinbing Wang for his support when I embarked on research projects.

I am also indebted to Ang Chen, Lin Zhong, Christopher M. Jermaine, Swarat Chaudhuri, Dan S. Wallach, Moshe Y. Vardi, T. S. Eugene Ng, and Nathan Dautenhahn for helping me navigate the early stages of my Ph.D. research and broadening my understanding of computer science.

My gratitude extends to my advisors at Meta, Aihua Liu and CJ Bell, who designed exciting internship projects that provided me with invaluable insights into the practical applications of programming language knowledge in the industry. I would also like to express my appreciation to my peers at Meta, Shuang Song, Donglei Fu, Justin Slepak, Jack Keller, and Mike Lui, for their guidance on software development, leveraging the skills I acquired during my PhD studies.

I am incredibly grateful to all my colleagues and friends at Rice. Special thanks

to Youyu Lu, Zhiwei Zhang, Yongze Yin, Yilei Fu, Cao Zhen, Weitao Wang, Jiarong Xin, Zhaozhuo Xu, Lechuan Li, Tiancheng Xu, Xuxin Tang, Kaixiong Zhou, Srdan Milakovic, Dimitrije Jankov, and Constantinos C. Chamzas for fostering a supportive and inclusive environment at Rice, where we shared both triumphs and challenges. I am deeply grateful for the unwavering support and friendship of Dongzhou Huang, Xinglin Wang, Yankai Wen, Sunny Huang, and Xiaoye Sun, who were instrumental in helping me overcome moments of doubt and providing clarity during my journey.

I extend my thanks to Alexis Le Glaunec, Zhifu Wang, Agnishom Chattopadhyay, Maki Yu, and Ziyuan Wen for their collaborative efforts as a team, enabling us to achieve significant research milestones. I would also like to express my gratitude to Ke Wu, Hongfei Ye, Zhitong Yan, Yujie Jiang, Peng Yang, Haixiao Wang, and Bochen Zhang for their companionship and friendship.

To my dear friends Yaowei Huang, Sheng Guan, Pinchen Xie, Yahao Dai, Yixuan Li, Jialu Zhuang, Wei Sun, and Xiaohan Chi, I am forever grateful for the enjoyable and relaxing conversations we shared. Furthermore, I would like to extend my appreciation to the friends I made through playing table tennis. Thank you, Zhiyong, Jiasui, Jeff, Huijun, Ziyi, Yingru, Jonathan, and Saadi, for the pleasure of being teammates and participating in regional and national games together.

Lastly, my sincere gratitude goes to my parents, Bo and Xianqing, for their encouragement that enabled me to undertake this academic journey that began in the small city of Wenzhou, continued in Shanghai, and eventually brought me to Houston. I also thank my extended family, Wanchuan, Lan, and Xiaoping, for their support.

Contents

Illustrations

Tables

Chapter 1

Introduction

Rapid advancements in technology have led to an enormous proliferation of streaming data, i.e., data that is generated in real-time and at high rates. Such data arise from diverse application domains. For example, in smart buildings [2, 3], IoT devices continuously monitor parameters like temperature, humidity, and energy consumption, providing real-time data that can be analyzed to optimize energy use, improve building safety, and enhance the productivity of the occupants. In healthcare monitoring [4, 5], wearable devices and smart medical equipment produce a continuous stream of patient data, including heart rate, blood pressure, and other crucial health parameters. Processing this real-time data can facilitate the early detection of potential health issues. Smart transportation systems [6, 7] employ GPS, cameras, and road sensors to generate real-time data for traffic management, route optimization, and accident prevention. Smart electricity grids [8, 9] utilize sensors and smart meters to stream data on electricity usage, enabling balance of supply and demand, and future capacity planning. In financial market analysis [10, 11], streaming data comes from real-time market data feeds, trading transactions, and social media sentiment analysis. Analysts can utilize this data to make immediate investment decisions and predict market trends. In the context of network traffic monitoring [12, 13], the continuous monitoring and analysis of network traffic data can ensure the smooth operation of networks, detect and respond to security incidents, and plan for future network capacity.

There are various proposals for specialized languages, compilers, and runtime

systems that handle the processing of streaming data. Relational database systems and SQL-based languages have been adapted to the streaming setting [14, 15, 16, 17, 18, 19]. Several systems have been developed for distributed processing of data streams, drawing on the dataflow model of computation [20, 21, 22]. Languages for detecting complex events in distributed systems, informed by regular expression theory and finite-state automata, have also been proposed [23, 24, 25, 26, 27, 28]. Synchronous dataflow programming languages have been employed for streaming computations [29, 30, 31, 32]. Additionally, various formalisms have been proposed for the runtime verification of reactive systems, many of which are based on Temporal Logic variants and their timed/quantitative extensions [33, 34, 35, 36, 37, 38]. A rich assortment of languages and systems for reactive programming also exists [39, 40, 41, 42], which focus on the development of event-driven and interactive applications such as web programming. Moreover, regular expression matching is widely used for recognizing patterns in data streams [43, 44], with various tools developed for matching regular patterns, encompassing both software engines [45, 46, 47] and hardware architectures [48, 49, 50, 51, 52, 53, 54].

Although the previously mentioned methods have demonstrated efficacy within their respective application domains, contemporary applications necessitate advanced language support to address the following challenges: (1) enabling high-level specifications for processing streaming data, (2) providing an efficient approach to identify regular patterns in a streaming setting, and (3) ensuring semantics-preserving parallelization of streaming computations.

1.1 Demand for High-level Specifications

Modern applications require further language support for the high-level specification of processing over data streams. The processing of data streams typically involves computations that integrate simple transformations, the detection of patterns, and streaming aggregations. For example, let us consider the data streams generated by sensors in a real-time health monitoring application (such as heart rhythm and brain activity monitoring). These signals contain noise from various sources. Moreover, they are mostly uneventful and interspersed with episodes of unusual activity that need to be identified and analyzed in a timely manner. Therefore, the monitoring application needs to perform a complex streaming computation that reduces the noise, identifies abnormal patterns in the signals, and summarizes the most important information.

One approach for specifying the processing of streaming data is to use a low-level imperative programming language such as C or C++. This approach quickly becomes difficult and error-prone, as the overall computation cannot be easily expressed in a modular way. The resulting program contains complex state-manipulating logic, and the code is highly entangled. For this reason, it is desirable to provide language support for assisting the programmer in specifying the application in a modular way by composing simpler computational primitives.

However, existing approaches do not provide all the necessary abstractions for specifying such complex computations in a natural and succinct way. For instance, while streaming SQL and related query languages [14, 15, 16, 17, 18, 19] focus on relational abstractions, they lack comprehensive support for computations that hinge on the temporal sequencing of events. Similarly, synchronous and reactive programming languages [29, 30, 31, 32, 55, 39, 40, 41, 42], despite providing dataflow abstractions, are less expressive when it comes to the modular specification of complex

temporal patterns. Additionally, although monitoring formalisms rooted in Temporal Logic [33, 34, 35, 36, 37] encompass some quantitative characteristics like timestamp comparisons, they offer inadequate support for crucial streaming computations such as aggregations and signal transformations.

1.2 Demand for Efficient Matching of Regular Patterns

The ever-increasing volume of streaming data across diverse domains presents the challenge of uncovering patterns embedded within this data. These patterns contain crucial insights, and their identification plays a vital role in a multitude of real-world applications such as financial market analysis [10, 11], network traffic monitoring [12, 13], healthcare [4, 5], telecommunications [56], and smart infrastructure management [2, 3, 8, 9]. In numerous real-world applications, many of these patterns emerge as regular patterns, and the matching of regular patterns has surfaced as an important technique in many stream processing systems [57, 58, 27, 24, 59, 60, 61].

Regular patterns, whether expressed using regular expressions or finite-state automata, are widespread across an extensive range of application domains beyond data stream processing. Regular patterns are employed in text analysis for searching, extracting, parsing, or transforming segments of text [62, 63, 64]. Additionally, they find applications in network security [43] for identifying signatures in network traffic that signal intrusions or other security issues, in bioinformatics [65, 66] for denoting DNA, RNA, or protein sequences, and in runtime verification [44, 67] for specifying safety properties. The extensive use of regular expressions in software projects further emphasizes their significance, as studies (e.g., [68]) indicate that 30%-40% of Java, JavaScript, and Python software projects rely on regex matching for various purposes.

Several different techniques have been developed for matching regular patterns,

with many relying on deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs) execution. In general, DFA-based techniques are faster, as processing an input element requires a single memory lookup, while NFA-based techniques are slower, as they involve extending multiple execution paths when processing an element. However, NFAs typically offer greater memory efficiency. In fact, for some NFAs, the minimal equivalent DFA would be exponentially larger [69].

Numerous applications require the processing of large and complex NFAs on real-time data streams. Both high-performance computing and battery-powered embedded applications prioritize energy and memory efficiency (in terms of memory capacity or chip footprint needed for a given NFA). NFA processing on general-purpose processors demands frequent, irregular, and unpredictable memory accesses, resulting in limited throughput and high power consumption on CPU and GPU architectures [70, 71, 72]. Field Programmable Gate Arrays (FPGAs) offer high-speed processing via hardware-level parallelism but often suffer from routing congestion-related bottlenecks [73, 74] and high power, area, and cost limitations that impede their use in mobile and embedded devices. Even digital application-specific integrated circuit (ASIC) accelerators face memory access bandwidth constraints that limit parallelism [75, 76]. The latest hardware technology addressing these challenges is in-memory architecture [54, 77, 53], which processes NFA transitions directly within memory using massive parallelism.

Classical regular expressions (regexes) involve operators for concatenation $\cdot$, nondeterministic choice $+$, and iteration (Kleene's star) *. They can be translated into NFAs whose size is linear in the size of the regex [78, 79]. However, regexes utilized in practice often include additional features that increase their succinctness. One such feature is *counting*, written as $r\{m, n\}$, which is also referred to as *bounded*

repetition. The pattern $r\{m, n\}$ expresses that the subpattern r is repeated between m and n times. This counting operator is ubiquitous in practical regex use cases. The naïve approach to handling counting operators is unfolding. For example, $r\{n, n\}$ is unfolded into $r \cdot r \cdots r$ (n-fold concatenation) and results in an NFA of size linear in n, potentially producing a DFA of size exponential in n. Since n can become quite large, dealing with bounded repetition is one of the main technical challenges for effectively utilizing hardware-based approaches to execute practical regular patterns. Existing in-memory NFA architectures employ this naïve unfolding method to handle counting, resulting in large memory and energy cost, which is inefficient.

1.3 Demand for Parallel Stream Processing with the Preservation of Sequential Semantics

Applications handling large-scale data streams have significant scalability requirements and can benefit from a multicore implementation of the streaming language or engine. For instance, in applications related to high-frequency stock trading, billions of quotes and transactions occur daily [80]. The complex analysis of such high-volume and high-velocity data necessitates a multicore implementation capable of meeting throughput requirements. Furthermore, applications dealing with streaming data typically demand strict correctness, ensuring in-order processing of input items. Many workloads, such as stock market price pattern analysis [26] and healthcare monitoring [81], demand a *semantics-preserving*, or *safe* parallel implementation, which, as emphasized in [1], must process items based on input order and produce results consistent with sequential implementations.

This thesis focuses on the parallel processing of data streams using multicore CPUs.

In this scenario, inter-thread communication is more reliable than in distributed systems, as messages or data items experience lower latency and a reduced likelihood of loss compared to network communication. To parallelize stream processing, programmers often express computation as a dataflow graph, with each node corresponding to a computation stage, where nodes communicate using FIFO channels. During compilation and deployment, this dataflow graph is mapped to physical cores and processes to achieve parallelism, thereby accelerating the overall streaming computation since multiple nodes can compute simultaneously.

Many tools have been developed to expose parallelism in stream processing based on dataflow graphs. However, they often lack adequate support for processing streaming data while ensuring safety in our context, i.e., the output of parallel computation should match that of sequential computation. For instance, distributed stream processing systems [82, 22, 20, 83, 84], such as Storm [20], cannot preserve sequential output order. Others, like Flink [83], utilize external APIs to reorder input item timestamps. Synchronous dataflow (SDF) languages and models [85, 30, 86, 31, 55] are valuable for implementing parallelism in streaming computations within the embedded software domain. However, SDF languages typically assume fixed item rates, implying that a given input item generates a fixed number of output items during computation. This property enables efficient scheduling determined at compile-time, but SDF languages inherently cannot express computations involving dynamic item rates, where an arbitrary number of output items may be generated from a given input item. A recent study [1] proposes a framework to ensure safe data parallelism for streaming computations with dynamic item rates. This framework offers multiple ordering strategies to preserve sequential semantics based on the use of sequence numbers. However, it is insufficient to maintain sequential semantics when

implementing computations with complex patterns of parallelism.

1.4 Contributions

This thesis aims to provide language support for real-time data processing through three approaches: (I) creating a language for specifying complex computations over real-time data streams, (II) developing software-hardware co-design for efficient detection of regular patterns in a streaming setting, and (III) designing a system for parallel stream processing with the preservation of sequential semantics.

The detailed contributions in each part are listed as follows:

Part I. We developed a novel language, called StreamQL, that simplifies the task of specifying complex streaming computations over data streams. Unlike existing approaches that primarily focus on streams as their basic object, such as the `Observable` in Rx [39], StreamQL uses *stream transformations* to define how input streams are converted into output streams. By focusing on stream transformations, StreamQL effectively integrates several valuable programming abstractions for stream processing, including: (1) *relational* constructs, such as filtering, mapping, aggregating, key-based partitioning, and windowing; (2) *dataflow* constructs, like streaming/serial and parallel composition; and (3) *temporal* constructs, which are inspired by Temporal Logic and regular expressions. Furthermore, the composition of stream transformations enables programmers to express streaming analyses as modular data queries, allowing for the flexible composition of computations.

We provided an implementation of StreamQL as a lightweight, high-throughput Java library. We compared our StreamQL implementation against popular

open-source streaming engines. The experiments show that our implementation consistently performs well when compared to these state-of-the-art streaming engines. In benchmarks with real-world applications, the throughput of StreamQL is consistently higher (up to 100 times) than these engines.

Part II. We proposed software and hardware co-design for integrating counting modules into state-of-the-art in-memory NFA architectures. This approach optimizes the memory and energy efficiency of NFA processing with counting while maintaining the performance advantages offered by in-memory architectures. By developing innovative methods to handle the counting construct, we can enhance the effectiveness of hardware-based approaches for executing regular patterns in various real-world applications.

In particular, we proposed a novel notion called counter-unambiguity to identify instances of bounded repetition that can be handled with a small amount of memory. We provided an efficient algorithm for analyzing counter-(un)ambiguity over regular expressions that arise in several application domains. We developed a compiler that translates POSIX-style regular expressions into high-level programming specifications used by the hardware, where the compiler first performs the static analysis for checking counter-(un)ambiguity and then leverages the analysis results to produce a low-level description of the automaton. We proposed a *hardware design* that augments the prior NFA-based CAMA architecture [53] with counter and bit vector modules. This architecture achieves substantial energy (up to 76%) and area (up to 58%) reductions compared to prior designs.

Part III. We proposed a novel ordering strategy for data items that leverages *signature*s, which consist of a series of sequence numbers that unambiguously

define the data processing order. We have developed a novel programming system using this signature-based approach, capable of preserving sequential semantics. This system enables programmers to effortlessly develop semantics-preserving parallel programs for expressing complex patterns of parallelism.

We implemented this system as a lightweight Rust library called ParaStream, which consists of a comprehensive set of operators for describing streaming computations executed by a node in the dataflow graph. Our algorithms for preserving sequential semantics are implemented on top of each node, allowing users to program their computations with semantics-preserving parallelism without the need of implementing low-level item reordering algorithms. We have compared this library to state-of-the-art engines for the efficient processing of data streams. The benchmarking results show that ParaStream consistently provides higher throughput than state-of-the-art tools in single-threaded settings and more substantial speedups with increasing degrees of parallelism.

The work in this thesis was done in close collaboration with my advisor, Konstantinos Mamouras, and other collaborators: particularly Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, and Kaiyuan Yang for works related to the matching of regular expressions. Qixuan and Kaiyuan lent their expertise to help complete the design of the hardware architecture, as presented in Chapter 4. Different chapters of this thesis correspond to several of my publications, as referenced [87, 88, 89].

My work is implemented in open-source tools:

- **StreamQL**[*] is a query language for efficient data stream processing.

[*] `https://ohyoukillkenny.github.io/source/streamql.html`

- **C-Ambiguity Checker**[†] is a tool that can provide a static analysis over regexes to determine whether a regex is counter-(un)ambiguous.

1.5 Thesis Overview

Chapter 2 provides a review of related literature and introduces the necessary notation and background. The technical part of this thesis has been split into three sections. Chapter 3 presents the design and the implementation of the StreamQL language. Chapter 4 shows our software-hardware co-design for efficient in-memory regular pattern matching. Chapter 5 presents the programming system used for parallel stream processing with the preservation of sequential semantics. Finally, Chapter 6 concludes this thesis.

[†] `https://ohyoukillkenny.github.io/source/regexchecker.html`

Chapter 2

Background and Related Works

This chapter surveys related work, introducing requisite notation and background. It covers languages and tools for stream processing, studies on regular expression matching, and research on maintaining sequential semantics in parallel stream processing.

2.1 Languages and Tools for Stream Processing

2.1.1 Streaming Database Systems

A large body of work exists on streaming database systems such as STREAM [90], Aurora [16], Borealis [18], CACQ [91], TelegraphCQ [17], Niagara [92], Gigascope [12], Nile [93], Microsoft's CEDR [94], and StreamInsight [95]. The languages supported by these database systems (for example, CQL [19]) are typically variants of SQL with additional streaming constructs for sliding windows. These languages are limited in their ability to perform computations that depend on the order of arrival of data items, such as detecting complex patterns.

2.1.2 Distributed Stream Processing Systems

Numerous distributed stream processing systems are based on the distributed dataflow model of computation, including S4 [96], IBM Streams [6], MapReduce Online [97], Storm [20], Summingbird [98], Heron [21], Naiad [82], Spark Streaming [22, 99], Flink [83], Google's MillWheel [100], Samza [101], and Beam [84]. Many of these systems,

such as Storm, Heron, and Samza, provide a low-level API for specifying a dataflow graph of operators, with each operator defined as an event-handling function. Others, including IBM Streams, Spark Streaming, Flink, and Beam, offer higher-level APIs for describing streaming computations. IBM Streams and Flink include specialized pattern detection operators based on regular expressions. All these systems aim to achieve high throughput, scalability, load balancing, load shedding, fault tolerance, and recovery. For instance, Spark Streaming utilizes micro-batches to ensure high throughput and fault recovery guarantees.

2.1.3 Complex Event Processing

Complex Event Processing (CEP) languages and tools focus on detecting complex patterns within event streams. These languages often rely on regular expressions [25] and are implemented using variants of finite-state automata [57, 58, 27, 24, 59, 60, 61] or evaluation trees [102, 103, 104, 105]. Several streaming engines, such as Trill [106], Esper [107], Siddhi [108], Flink [28], Oracle Stream Analytics [109], and IBM Streams [6], offer specialized operators or extensions for CEP. A typical issue with CEP implementations is the need to explore all possible input stream parse trees that match a regular expression, which can lead to an exponential blowup in memory requirements in the worst case.

2.1.4 Lightweight Streaming Engines

Several lightweight streaming engines are implemented as libraries within a general-purpose host language, enabling easy integration with other systems and allowing existing code to be repurposed for application-specific tasks. Microsoft's Trill [106] is a high-performance streaming library that utilizes a batched-columnar data rep-

resentation and dynamic compilation. Trill accommodates out-of-order events and offers extensions for pattern matching and signal processing [110, 57, 111]. Esper [107] and Siddhi [108] are lightweight engines for CEP and streaming analytics, providing a rich set of operators, including SQL-based constructs, windows, and pattern matching. Java Stream [112] and Stream Fusion [113] deliver a simpler streaming API for processing static data collections. NetQRE [114] and StreamQRE [115] integrate unambiguous regular expressions with quantitative calculations and other streaming constructs like streaming composition [116]. The core of these two languages is related to transducers with registers capable of holding values [117, 118, 119] and other related models [120]. InfluxDB [121] is a database system optimized for time-series data and implements two query languages: (1) InfluxQL, based on SQL syntax, and (2) Flux [122], featuring a more functional syntax.

2.1.5 Reactive Programming

The survey paper [123] explores various approaches to reactive programming. Functional reactive programming (FRP) focuses on the transformation of time-varying values (signals), with early representative languages including Fran [40] and Yampa [124]. Several subsequent frameworks embed FRP in imperative languages, such as Flapjax [125], Frappé [41], and Scala.React [42]. FRP has primarily been used in developing event-driven and interactive applications like GUIs. Elm [126] is a practical FRP language designed for the easy creation of responsive GUIs. Libraries such as Rx [127, 39], Reactor [128], and Akka Streams [129] share similarities with FRP languages but deal with event streams instead of signals. These libraries present data streams as push-based collections (e.g., `Observable` in Rx) and provide APIs for transforming these streams.

2.1.6 Signal Processing on Streams

The WaveScope project [130, 131] highlights the need to combine event-stream processing with signal processing for applications that make use of sensor-generated data streams. TrillDSP [111] extends Trill with signal processing functionality. LifeStream [132] is a stream processing engine for physiological data, offering comprehensive temporal query language support for signal processing.

2.2 Matching of Regular Expressions

2.2.1 Regular Expression with Bounded Repetition

Regular expressions, often abbreviated as *regex* or *regexp*, represent a widely used formalism for describing regular patterns. Classical regular expressions involve constructs for nondeterministic choice $r_1 | r_2$, concatenation $r_1 \cdot r_2$, and Kleene's star r^* (repetition of r zero or more times). They can be translated into NFAs whose size is linear in the size of the regex [78, 79].

In practice, the syntax of regular expressions is often extended with more features for convenience and succinctness, such as character classes for describing sets of letters/symbols (e.g., $[ab]$ and $[0{-}9]$), the construct $r?$ for indicating that the pattern r is optional, and Kleene's plus r^+ (repetition of r at least once). The construct of bounded repetition (i.e., counting), which is written as $r\{m, n\}$ and describes the repetition of r from m to n times, can be translated using concatenation and ? but makes regular expressions exponentially more succinct. In practice, $r\{n\}$ is written as the abbreviation for $r\{n, n\}$. The expression $r\{n, \} = r\{n\}r^*$ describes the repetition of r at least n times. The construct of bounded repetition is ubiquitous in practical use cases of regexes. For example, it is extremely common in datasets for network

intrusion detection [133, 134] and motif search in biological sequences [135, 65]. The naïve approach for dealing with bounded repetition is to rewrite it by *unfolding*. For example, $r\{n\}$ is unfolded into $r \cdot r \cdots r$ (n-fold concatenation) and results in an NFA of size linear in n (and therefore can produce a DFA of size exponential in n).

Formalization of Regular Expressions with Bounded Repetition

Let Σ be a finite alphabet. This thesis considers a set of regular expressions, which we call Regex, over Σ. Regex is the smallest set that satisfies the following properties:

1. Regex contains the expression ε (regex that recognizes the empty string).
2. Regex contains every predicate σ over the alphabet (i.e., $\sigma \subseteq \Sigma$).
3. For every $r, r_1, r_2 \in \mathsf{Regex}$, $r_1 \cdot r_2 \in \mathsf{Regex}$ (concatenation), $r_1 | r_2 \in \mathsf{Regex}$ (nondeterministic choice) and $r^* \in \mathsf{Regex}$ (Kleene's star).
4. For every $r \in \mathsf{Regex}$ and all integers m, n with $0 \leq m \leq n$, $r\{m, n\} \in \mathsf{Regex}$.

The concatenation symbol is sometimes omitted, i.e., $r_1 \cdot r_2$ sometimes is written as $r_1 r_2$. The interpretation of a regex r is a language $\mathcal{L}(r) \subseteq \Sigma^*$, which is defined in the standard way.

Notation for Character Classes

A predicate over the alphabet is also called a *character class*. The predicate Σ contains all symbols in the alphabet. When we use a symbol $a \in \Sigma$ in a regex, it should be understood as the singleton predicate $\{a\} \subseteq \Sigma$. The notation $[a_1 \ldots a_n]$ is used for representing the predicate $\{a_1, \ldots, a_n\} \subseteq \Sigma$. We write $[\hat{} a_1 \ldots a_n]$ for the predicate $\Sigma \setminus \{a_1, \ldots, a_n\}$ that contains all symbols except for $a_1, \ldots, a_n$.

2.2.2 Regular Expression Matching on Software

There are several approaches for implementing the matching of regular expressions on software (i.e., CPU). The regex matching engines of many programming languages such as Java, .NET, Python, and JavaScript use backtracking search. For some regexes, these engines need time that is exponential in the size of the input text. For other regexes, backtracking engines may need time that is polynomial in the length of the input text, but the polynomial has degree at least 2. This is a lot worse than Thompson's algorithm [78], which has time complexity $O(m \cdot n)$, where m is the size of the regular expression and n is the size of the input text. The benefit of backtracking is the simplicity of its implementation and the ease with which advanced features (e.g., lookaround [136] and backreferences [137]) can be added.

Many modern regex engines are based on the classical theory of automata. They employ DFAs or NFAs, or both [63, 45, 47]. DFA-based algorithms perform a memory lookup when receiving an input symbol to execute the transition of the automaton. This makes them fast, because they need $O(1)$ time per symbol. One potential problem is that the representation of the DFA might require a large amount of memory. For many patterns that arise in practice, their encoding as NFAs can be substantially more compact that their encoding as DFAs. In fact, it is known that there are families of patterns for which minimal DFAs are exponentially larger than equivalent NFAs [69]. The downside with NFAs is that their execution is less efficient in terms of time complexity. NFA execution involves maintaining a set of active states, which represent several possible execution paths, and performing at every step a transition for each one of the currently active states. So, for every step, this may involve work that is proportional to the number of states of the NFA.

2.2.3 Regular Expression Matching on Hardware

Software-based implementations of regular pattern matching often explore this trade-off between time- and memory-efficiency that we described in the previous subsection. There are also hardware-based implementations that make use of the inherent parallelism that is available in hardware, where circuit elements compute independently and in parallel. There are several proposals that use field-programmable gate arrays (FPGAs) to support the matching of regexes [138, 139, 140, 73, 74, 141, 142, 143]. A number of digital application-specific integrated circuit (ASIC) accelerators [52] have been designed to achieve high throughput for regex matching. The IBM RegX [76] accelerator expands on the concept of representing regexes with compressed DFAs [144, 145, 146] and its parallelized architecture enhances performance on large workloads. The Automata Processor (AP), introduced in [54], is a reconfigurable ASIC hardware based on bit-parallelism [147] that simulates NFAs in parallel. SparseAP [148] enables AP to efficiently execute large-scale applications. AP can support numerous regexes found in real-life applications [70, 149], but provides limited support for regexes with counting (when upper bounds exceed 512, they are considered unbounded [150]). Other significant ASIC works are based on the Aho-Corasick algorithm [151], including [51], HAWK [75], and HARE [152]. Moreover, there are several works that implement regex matching algorithms on GPUs [153, 154, 48, 155, 156]. Hardware offers the memory-efficiency of NFAs without a penalty in execution time, since NFA transitions from active states are explored in parallel in one step.

2.2.4 Automata With Counters

Previous studies have proposed automata with counters for efficiently handling regular expressions with bounded repetitions (i.e., counting). XFA extends traditional DFA

with counters to reduce memory requirements when representing regular expressions with bounded repetitions [157]. Similarly, counting-NFA [144] extends NFA for dealing with counting in regular expressions. The determinization of a class of counter automata is considered in [158]. This determinization produces smaller automata than standard determinization into DFAs. The work of [158] is extended in [159] and the CA regex engine, which handles bounded repetition using a specialized algorithm, is presented. The CA engine uses a class of automata with counters, called "counting automata" (CAs), which are nondeterministic, to represent patterns. CAs are converted into deterministic "counting-set automata" (CsAs). Using CsAs, the work of [159] develops an algorithm to efficiently handle certain cases of bounded repetition. One issue with the approach of [159] is that the automata underlying the matching algorithm may over-approximate the language of the pattern, potentially resulting in false matches.

2.3 Semantics-preserving Parallel Stream Processing

2.3.1 Classical Parallel Programming Models

Several classical parallel programming models have been developed to parallelize serial programs for general applications. OpenMP [160] uses directives like `critical` and `atomic` to design safe programs. Cilk [161, 162], a parallel C-like language, can avoid data races with specific data structures called *hyperobjects*. X10 [163], a Java-based language, requires static analysis to identify parallel tasks that may lead to data races. The MPI library [164] offers communication and distribution primitives for programs running on large clusters.

2.3.2 Kahn Process Networks

Dataflow programming models are extensively utilized in parallel computing and distributed systems, wherein a program is decomposed into nodes that communicate exclusively through First-In-First-Out (FIFO) channels. The history of the dataflow model can be traced back to Petri nets [165, 166] followed by several pioneering studies on the dataflow model [167, 168]. Kahn Process Networks (KPN) [169] is one of the prominent dataflow programming models. KPN operates by decomposing the computation into independent processes or nodes that interact solely through FIFO channels. When a KPN node is scheduled for execution, it becomes challenging to ascertain the precise number of data items it will read from or write to its input or output channels. Consequently, determining buffer sizes and scheduling the activation of nodes cannot be statically determined. Moreover, there is no guarantee that a KPN program can be executed using finite memory.

2.3.3 Synchronous Dataflow Programming Models

In the pursuit of achieving static predictability of boundedness, researchers have explored various models and languages. One such variant is the synchronous dataflow programming model, also known as SDF [85], which effectively is the restriction of KPN. In SDF, the programmer specifies, for each node, the static number of data items the node reads or writes on each of its input or output channels during activation.

The SDF models are useful for explicitly identifying the parallelism present in streaming computations that arise in the embedded software domain, such as signal processing [85] and embedded controller design [30, 86, 31, 170]. The StreamIt language, in particular, provides a general framework for streaming signal processing with efficient execution on multicore architectures [55].

2.3.4 Actors and Active Objects

Various languages employ actors and active objects for parallel computation. Actors [171] are single-threaded entities that communicate asynchronously, while active objects [172] merge actors and object-oriented programming concepts, using asynchronous method invocation for communication. Erlang [173] is a prominent actor language known for massive parallelism and industry use. Rebeca [174, 175] is an actor-based language for modeling and verifying concurrent and distributed systems. There are also other languages based on the use of actors, which includes ABS [176], ASP [177], Encore [178], etc. The actor community has mitigated the inherited nondeterminism of actors and associated bugs through advanced debugging tools [179, 180, 181, 182], while Lingua Franca [183], a reactor-based language, augments mainstream languages with a concurrency model for exploiting parallelism without nondeterminism.

2.3.5 Algorithmic Skeletons

Algorithmic skeletons [184] abstract common patterns of parallel computation. Using skeletons, programmers can express parallel computations using a composition language that assembles basic sequential blocks. Classic parallel programming patterns, like map, reduce, pipeline, farm, and divide and conquer are used for skeleton composition. Libraries such as ASSIST [185], Calcium [186], Eden [187], Muskel [188], P^3L [189], Skandium [190], etc., offer various algorithmic skeletons for parallel programming. The determinism of the parallel computation is not the primary focus in the design of the algorithmic skeletons. However, several skeleton programs are inherently deterministic due to independent sequential skeletons and composition operations.

2.3.6 Lightweight Libraries for Parallel Stream Processing

There are several lightweight parallel streaming libraries that are implemented as libraries within a general-purpose host language. Microsoft's Trill [106] is a high-performance streaming library that employs a batched-columnar data representation and dynamic compilation. Trill provides streaming generalizations of the classic MapReduce operations with temporal support to allow parallel stream processing. Inspired by Trill's columnar store and bit-vector design, StreamBox [191, 192, 193] exploits the parallelism and memory hierarchy of modern multicore hardware with more efficient data structures to support high-performance stream processing. Java Stream [112] and PLINQ [194] provide streaming APIs. PLINQ provides mechanisms to reorder the output data generated by multiple worker threads during parallel data processing.

2.3.7 Correctness of Parallel Stream Processing

To ensure the correctness for parallel stream processing, many existing works focus on the testing of batch processing programs under the MapReduce framework [195, 196, 197] and of general dataflow or stream-processing programs [198, 199]. There are also language-based approaches that enforce correct (i.e., semantics-preserving) parallelization in stream processing programs [1, 200, 201, 202]. In particular, [1] has presented a runtime system that is capable to preserve the sequential semantics in the presence of operators that can be stateful and have dynamic item rates. Moreover, [202] proposes dependency-guided synchronization (DGS), which is an alternative programming model for streaming computations with complex synchronization requirements. DGS requires users to specify the dependency relation between input events and decomposes the input stream using the fork-join operations [203, 204].

Chapter 3

Language Design for Stream Processing

3.1 Motivation

The emergence of new technologies, notably the Internet of Things (IoT), has led to a significant increase in the amount of streaming data. This type of data is created in real time and at high rates. Such data arise in various application domains, such as smart buildings [2, 3], healthcare monitoring [4, 5], smart transportation [6, 7], smart electricity grids [8, 9], financial market analysis [10, 11], telecommunications [56], and network traffic monitoring [12, 13].

There are various proposals for specialized languages, compilers, and runtime systems that deal with the processing of streaming data. Relational database systems and SQL-based languages have been adapted to handle streaming data [14, 15, 16, 17, 18, 19]. Several systems have been designed for the distributed processing of data streams using the dataflow model of computation [20, 21, 22]. Languages for detecting complex events in streaming data, which draw on the theory of regular expressions and finite-state automata, have also been proposed [23, 24, 25, 26]. Synchronous dataflow programming languages [29, 30, 31, 32] have been used for streaming computations in embedded systems. Several formalisms for the runtime verification of reactive systems have been proposed, many of which use variants of Temporal Logic and its timed/quantitative extensions [33, 34, 35, 36, 37]. Finally, there exists a wide range of languages and systems for reactive programming [39, 40, 41, 42], which focus on

creating event-driven and interactive applications.

While the approaches mentioned above have proven effective in their respective fields of application, modern applications demand further language support for the high-level specification of processing over data streams. The processing of such data typically involves computations that integrate simple transformations, pattern detection, and streaming aggregations. For instance, in a real-time health monitoring application, data streams from sensors, such as heart rhythm and brain activity, often contain noise and uneventful data mixed with unusual activity. Therefore, the monitoring application needs to perform complex streaming computations to reduce noise, identify abnormal patterns, and summarize the most important information.

Using a low-level imperative programming language like C or C++ for processing streaming data can be complex and prone to errors, as it is challenging to express the overall computation in a modular fashion. The resulting program contains complex state-manipulating logic and the code is highly entangled. Hence, it is desirable to provide language support that helps programmers to specify the computation in a modular way by composing simpler computational primitives. However, existing approaches do not provide all the necessary abstractions for specifying such complex computations in a natural and succinct way. For example, streaming SQL and related query languages focus on relational abstractions, but they provide limited support for computations that rely on the temporal sequencing of events. The synchronous and reactive languages offer dataflow abstractions, but they are less suitable for the modular specification of complex temporal patterns. Monitoring formalisms based on Temporal Logic include quantitative features like timestamp comparisons and simple value thresholds, but they offer little support for aggregations and signal transformations.

3.2 Contributions

This thesis presents StreamQL (Streaming Query Language), a language designed to simplify complex streaming computations over real-time data. This proposed language diverges from existing proposals, in which the fundamental component is the data stream such as the `Observable` in Rx [39] and the `IStreamable` in Trill [106]. Instead, the basic object of StreamQL is the **stream transformation**, which describes how each input stream is transformed into an output stream. StreamQL provides a novel fusion of several useful programming abstractions for stream processing: (1) relational constructs (such as filtering, mapping, aggregating, key-based partitioning, and windowing), (2) dataflow constructs (such as streaming/serial and parallel composition), and (3) temporal constructs that are inspired from Temporal Logic and regular expressions. Beyond providing these abstractions, StreamQL empowers programmers with a modular approach for specifying streaming analyses. The language constructs offered in StreamQL are freely composable, allowing a flexible combination of different computations.

In StreamQL, a stream transformation is captured syntactically with a *query*. We classify queries according to their *input/output type* in order to guarantee that composite queries (i.e., queries that result from the composition of simpler queries) are well-formed. We write `f` : `Q`(A, B) to indicate that the query `f` processes an input stream with items of type A and produces an output stream with items of type B. A stream is typically viewed as an unbounded sequence of data items (elements). A key feature of StreamQL is that it generalizes this notion of a stream by allowing the occurrence of a distinguished symbol $\square$, called *end-of-stream marker*, that signals the termination of the stream. This is useful not only because there are certain streams that indeed terminate (e.g., when reading lines from a text file), but more importantly

because it allows us to decompose unbounded streams into finite regions: each finite region can be viewed as a stream that eventually terminates. Such decompositions of streams are essential for the modular description of complex streaming computations. A key design feature of StreamQL is that a query can *halt* (terminate), even before the input stream has terminated. After a query has halted, then our language allows the computation to proceed according to some other query, thus varying the computation over time. This novel feature of StreamQL enhances modularity by enabling the unrestricted composition of temporal and dataflow/relational operators.

StreamQL has an expressive set of combinators for describing common stream processing primitives, as well as rich forms of composition. The primitive queries `map`, `filter`, `reduce` and `aggr` describe basic streaming operators for transforming, filtering, and aggregating streams. The combinator `groupBy` supports the key-based partitioning of a stream and independent computation over disjoint sub-streams, which is similar to the Group-By construct in database query languages. The windowing combinators `tWindow` (tumbling) and `sWindow` (sliding) facilitate the specification of computations that operate on finite spans of an unbounded data stream. The combinators $\gg$ (streaming/serial composition) and `par` (parallel composition) allow the programmer to describe a complex computation as a directed acyclic graph of independent tasks, which facilitates modular specification and exposes pipeline and task parallelism. The atomic queries `takeUntil`, `skipUntil` and `search` are used to identify simple single-event patterns in a stream. They are inspired from the *Until* connective of Temporal Logic. The combinators `seq` (temporal sequencing) and `iter` (temporal iteration) are useful for describing time-varying analyses and detecting complex temporal patterns. The constructs `seq` and `iter` can be viewed as stream-transforming analogs of concatenation and Kleene's star from regular expressions.

The StreamQL language has a formal denotational semantics. A query `f` : `Q`(A, B) represents a monotone function $A^* \cdot \{\varepsilon, \square\} \to B^* \cdot \{\varepsilon, \square\}$, where * is Kleene's star, ε is the empty string, and $\cdot$ is string concatenation. The monotonicity requirement captures a key requirement of streaming computation: an output item cannot be retracted after it has been emitted to the output. Every combinator of StreamQL has a denotational semantic analog, which provides unambiguous meaning for the entire language and thus validates the language design.

We provided an implementation of StreamQL as a lightweight Java library. We use an explicit mechanism to *reset* the streaming computation, which allows us to reuse the allocated memory when performing operations that involve stream decomposition. In addition to the core combinators, the implementation provides support for aggregation (e.g., median and general percentiles), efficient algorithms for sliding windows, signal-processing primitives such as FFT (Fast Fourier Transform), FIR (Finite Impulse Response) filters, IIR (Infinite Impulse Response) filters, and several common stream processing idioms. We have used the library to specify real-world streaming applications for health monitoring.

We compare our StreamQL implementation against three popular open-source streaming engines: RxJava [205], Reactor [128], and Siddhi [108]. The experiments show that our implementation consistently performs well when compared to these state-of-the-art streaming engines. In benchmarks with real-world applications, the throughput of StreamQL is 1.1–10 times higher than RxJava, 1.2–20 times higher than Reactor, and 5–100 times higher than Siddhi.

The main contribution of this work lies in defining language abstractions for data stream processing that balance the following desirable characteristics:

- They provide clear formal semantics.

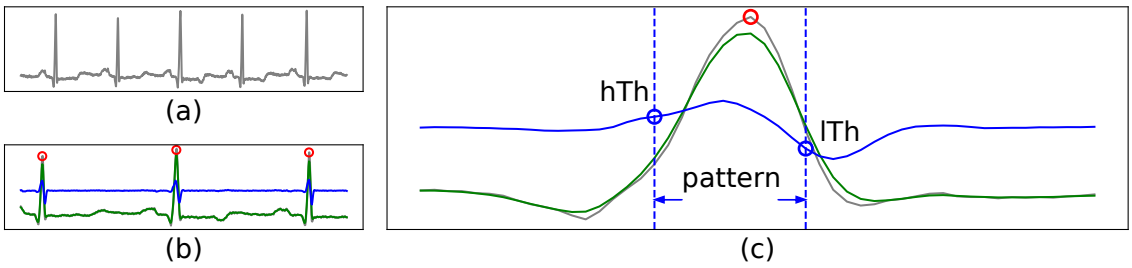


Figure 3.1 : (a) Electrocardiogram or ECG, (b) ECG with annotated signal peaks, (c) Pattern for peak detection.

- They give rise to an expressive and compositional language.
- They enable an efficient lightweight implementation.

The key design choice of StreamQL is to base the language on stream transformations that can potentially halt (instead of nested streams) and combinators on them. With respect to the implementation, a key idea is the introduction of an execution model that is essentially a stream transducer [206] that receives a special control signal for resetting its internal state. We have used this model to provide an efficient implementation of the language that avoids common sources of computational overheads that are present in related streaming languages.

3.3 Overview of StreamQL

To provide an overview of StreamQL, we will examine the processing of an electrocardiogram (ECG), which is a cardiac signal from a patient. We will particularly focus on the problem of peak detection in the ECG, which equates to detecting heartbeats. This problem has gained considerable attention in the realm of biomedical engineering [207, 208], as it underpins numerous analyses performed on cardiac data.

Figure 3.1(a) shows part of an ECG, which is the electrical cardiac signal recorded

on the surface of the skin near the heart. The horizontal axis is time and the vertical axis is voltage. A simple but effective procedure for detecting the peaks consists of three stages: (1) smoothing the signal to eliminate high-frequency noise, (2) taking the derivative of the smoothed signal to calculate the slope, and (3) finding the peaks using both the raw measurements and the derivatives.

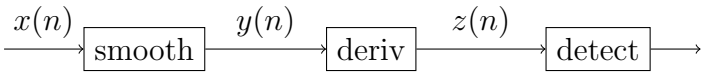


Figure 3.1(b) shows a short snippet (about 3 seconds) of an ECG signal, where the gray line corresponds to the input time series $x(n)$, the green line is the smoothed data $y(n) = (x(n-2) + 2x(n-1) + 4x(n) + 2x(n+1) + x(n+2))/10$, and the blue line is the derivative $z(n) = y(n) - y(n-1)$. A straightforward algorithm for detecting the peaks is to find the first occurrence (let us say at time i) where the derivative $z(n)$ exceeds a pre-defined threshold `hTh`, followed by the first occurrence after i (let us say at time j) where the derivative $z(n)$ becomes less than a threshold `lTh`. Time point i is located on the ascending slope towards the peak, and time point j is located on the descending slope after the peak. So, the exact peak location can be found by searching for the maximum value of the original $x(n)$ time series in the interval from i to j. This pattern is illustrated in Figure 3.1(c). Every time a peak is identified, this detection procedure is reset and repeated.

This motivating example shows that the detection of complex patterns requires the transformation of the data stream (e.g., smoothing and differentiation) to enrich it with extra information. For this reason, the basic concept in the design of StreamQL is the *stream transformation*, which specifies how an input stream is transformed into an output stream. A *query* is a syntactic description of a stream transformation. Every query `f` has a *type* `Q`(A, B), where A is the type of input data items, and B is

Table 3.1 : Map, Filter, Aggregate, and Reduce.

input:	$\langle 2.5, 1\rangle$	$\langle 0.8, 2\rangle$	$\langle 3.5, 3\rangle$	$\langle 0.9, 4\rangle$	$\square$
`f` output:	5.0	1.6	7.0	1.8	$\square$
`g` output:	$\langle 2.5, 1\rangle$		$\langle 3.5, 3\rangle$		$\square$
`h` output:	2.5	3.3	6.8	7.7	$\square$
`k` output:					7.7 $\square$

the type of the output data items. We write $\texttt{f} : \texttt{Q}(A, B)$ to indicate that `f` is of type $\texttt{Q}(A, B)$.

A stream is typically viewed as an unbounded sequence of data items. We consider here a generalization of this notion of streams by assuming that the stream can potentially contain an occurrence of a special $\square$ symbol, called *end-of-stream marker*, that signals the end of the stream. We say that a stream is *terminated* if it ends with $\square$. As we will see later, the introduction of the end-of-stream marker allows us to define stream transformations that operate only on finite parts of the stream, which is useful for the modular specification of complex streaming computations such as time-varying analyses.

We will proceed to present the basic programming constructs of the StreamQL language. We will start with some simple primitives, and we will gradually build up toward the more complex combinators (i.e., composition constructs) of the language. Finally, we will conclude this section with a complete description of the ECG peak detection algorithm.

Primitive Constructs: Map, Filter, Aggregate, and Reduce

Suppose that the input stream is a real-valued discrete-time signal. The type of the input data items is a record type $\texttt{VT} = \{\texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}\}$, where `V` is the type of scalar values (e.g., real numbers) and `T` is the type of time points (e.g., natural

numbers). The *map* query $\texttt{f} = \texttt{map}(\texttt{x -> } 2 \cdot \texttt{x.val})$ has type $\texttt{Q}(\texttt{VT}, \texttt{V})$ and represents the transformation that outputs the double of the value of each item. The argument $\texttt{x -> } 2 \cdot \texttt{x.val}$ is a lambda expression that defines a function of type $\texttt{VT} \to \texttt{V}$. The *filter* query $\texttt{g} = \texttt{filter}(\texttt{x -> x.val} \geq 2.0)$, of type $\texttt{Q}(\texttt{VT}, \texttt{VT})$, filters out those items with a value less than 2.0 and keeps the rest. The lambda expression $\texttt{x -> x.val} \geq 2.0$ is a predicate on VT. The *aggregation* query $\texttt{h} = \texttt{aggr}(0.0, (\texttt{x}, \texttt{y}) \texttt{ -> x} + \texttt{y.val}) : \texttt{Q}(\texttt{VT}, \texttt{V})$ represents the running sum of the values in the input stream. The first argument $0.0 : \texttt{V}$ is the initial aggregate value, and the second argument is a binary function of type $\texttt{V} \times \texttt{VT} \to \texttt{V}$ that specifies how to aggregate each input data item. The *reduce* query $\texttt{k} = \texttt{reduce}(0.0, (\texttt{x}, \texttt{y}) \texttt{ -> x} + \texttt{y.val})$, of type $\texttt{Q}(\texttt{VT}, \texttt{V})$, is similar to the running aggregation query h, with the difference that it only emits the total aggregate when the input stream terminates. Table 3.1 shows the execution of the queries f, g, h, k, where time progresses in the left-to-right direction. For a function $\texttt{op} : A \times A \to A$, we also consider the variants $\texttt{aggr}(\texttt{op}), \texttt{reduce}(\texttt{op}) : \texttt{Q}(A, A)$, which do not need an initial aggregate (the first item of the input serves this purpose). The most general variants take a function $\texttt{init} : A \to B$ for initialization (using the first item of the input) and an aggregation function $\texttt{op} : B \times A \to B$.

Key-based Partitioning

Let us consider an input stream with items of type $\texttt{IV} = \{\texttt{id} : \texttt{ID}, \texttt{val} : \texttt{V}\}$, where ID is a type of the identifier. Suppose that we have written a query $\texttt{f} : \texttt{Q}(\texttt{IV}, B)$ that computes an aggregate of items with a fixed identifier, i.e. under the assumption that all the items of the input stream have the same identifier. Then, to compute this aggregate across all identifiers, the most natural way is to partition the input stream by a key, the identifier field id in this case, and supply the corresponding

Table 3.2 : Key-based partitioning.

input:	$\langle a, 3\rangle$	$\langle b, 5\rangle$	$\langle a, 1\rangle$	$\langle c, 2\rangle$	$\langle c, 1\rangle$	$\langle a, 4\rangle$	□
group a:	$\langle a, 3\rangle$		$\langle a, 1\rangle$			$\langle a, 4\rangle$	□
group b:		$\langle b, 5\rangle$					□
group c:				$\langle c, 2\rangle$	$\langle c, 1\rangle$		□
`g` output:	3	5	4	2	3	8	□

projected sub-stream to a copy of `f`. This construct is called *key-based partitioning* and it is described by the query $\texttt{g} = \texttt{groupBy}(\texttt{x -> x.id}, \texttt{f}) : \texttt{Q}(\texttt{IV}, B)$. The first argument `x -> x.id` is a function of type $\texttt{IV} \to \texttt{ID}$ that specifies the partitioning key, and `f` describes the computation that will be independently performed on each sub-stream. If we choose $\texttt{f} = \texttt{aggr}(0, (\texttt{x}, \texttt{y}) \texttt{ -> x} + \texttt{y.val}) : \texttt{Q}(\texttt{IV}, \texttt{V})$ to be a running sum, then the query $\texttt{g} = \texttt{groupBy}(\texttt{x -> x.id}, \texttt{f}) : \texttt{Q}(\texttt{IV}, \texttt{V})$ performs the computation shown in Table 3.2.

Constructs: Tumbling & Sliding Windows

The so-called windowing constructs are used to partition an unbounded stream into finite fragments called windows and perform computations on each one of them independently. The *tumbling window* combinator splits the stream into contiguous non-overlapping regions. For a query $\texttt{f} : \texttt{Q}(A, B)$ and a natural number $n \geq 1$, the query $\texttt{tWindow}(n, \texttt{f})$ applies `f` to tumbling windows of size n. The *sliding window* combinator splits the stream into overlapping regions. For a query $\texttt{f} : \texttt{Q}(A, B)$ and natural numbers n, s with $1 \leq s < n$, the query $\texttt{sWindow}(n, s, \texttt{f})$ applies `f` to windows of size n with a new window starting every s items. Let us consider now the query $\texttt{f} = \texttt{reduce}(0, (\texttt{x}, \texttt{y}) \texttt{ -> x} + \texttt{y}) : \texttt{Q}(\texttt{V}, \texttt{V})$, which calculates the total sum of a terminated stream. Table 3.3 illustrates `tWindow` and `sWindow`. We also provide variants of the windowing constructs that allow the programmer to specify a function

Table 3.3 : Tumbling and sliding windows.

input:	1	2	3	4	5	6	7	8	□
`tWindow`$(2, \mathtt{f})$ output:		3		7		11		15	□
`tWindow`$(3, \mathtt{f})$ output:			6			15			□
`sWindow`$(2, 1, \mathtt{f})$ output:		3	5	7	9	11	13	15	□
`sWindow`$(3, 1, \mathtt{f})$ output:			6	9	12	15	18	21	□
`sWindow`$(3, 2, \mathtt{f})$ output:			6		12		18		□

Table 3.4 : Streaming (serial) composition.

input:	$\langle 2.5, 1\rangle$	$\langle 0.8, 2\rangle$	$\langle 3.5, 3\rangle$	$\langle 0.9, 4\rangle$	□
`f` output:	2.5	0.8	3.5	0.9	□
`f` $\gg$ `g` output:	2.5	2.5	3.5	3.5	□

`op` : $A^n \to B$ to summarize the contents of a window of size n, as in `tWindow`$(n, \mathtt{op})$ and `sWindow`$(n, s, \mathtt{op})$. For example, the query `sWindow`$(3, 1, (\mathtt{x}, \mathtt{y}, \mathtt{z})$ `->` $(\mathtt{x} + \mathtt{y} + \mathtt{z})/3)$ computes the sliding (moving) average over windows of size 3.

Streaming/Serial Composition

A natural construct for streaming computation is to compose queries `f` : `Q`(A, B) and `g` : `Q`(B, C) so that the output items produced by `f` are supplied as input to `g`. This is denoted by `pipeline`$(\mathtt{f}, \mathtt{g})$: `Q`(A, C), which we abbreviate as `f` $\gg$ `g`. We call this query the *streaming* or *serial composition* of `f` and `g`. This construct generalizes to more than two arguments. It is useful for setting up a complex computation as a pipeline of stages. Consider the queries `f` = `map`(`x->x.val`) : `Q`(`VT`, `V`) and `g` = `aggr`(`max`) : `Q`(`V`, `V`). The query `f` $\gg$ `g` : `Q`(`VT`, `V`) computes the running maximum (see Table 3.4).

Table 3.5 : Parallel composition.

input:	10	20	30	40	50	□
`f` output:	10	30	60	100	150	□
`g` output:	1	2	3	4	5	□
`par(f,g)` output:	10 1	30 2	60 3	100 4	150 5	□
`h` output:	10	15	20	25	30	□

Parallel Composition

We introduce a construct for executing multiple queries in parallel on the same input stream and combining their results. For queries `f` and `g` of type `Q`(A, B), the query `par`$(\texttt{f}, \texttt{g})$: `Q`(A, B) describes the following computation: The input stream is duplicated with one copy sent to `f` and one copy sent to `g`. The queries `f` and `g` compute in parallel, and their outputs are merged (specifically, interleaved) to produce the final output. Using the running sum query `f` = `aggr`$(+)$ and the running count query `g` = `aggr`$(0, (\texttt{x}, \texttt{y})$ `->` $\texttt{x} + 1)$, both of type `Q`$(\texttt{V}, \texttt{V})$, the query `h` = `par`$(\texttt{f}, \texttt{g}) \gg$ `tWindow`$(2, (\texttt{x}, \texttt{y})$ `->` $\texttt{x}/\texttt{y}) :$ `Q`$(\texttt{V}, \texttt{V})$ computes the running average (see Table 3.5). The `par` construct generalizes to several arguments.

Temporal Constructs

As mentioned before, the end-of-stream marker □ indicates the end of a stream. When a query emits □ to the output we say that it *halts*, because it cannot produce any more output. All the query examples that we have seen so far have the property that they halt exactly when they encounter □ in the input stream. By lifting this restriction we can support queries that can halt early. This is useful (1) for varying a streaming computation as time progresses and (2) for detecting complex temporal patterns.

Table 3.6 : Take, Skip, Ignore, and Search.

input:	1	2	3	4	5	□
takeUntil(x -> x ≥ 4) output:	1	2	3	4 □		
take(3) output:	1	2	3 □			
skipUntil(x -> x ≥ 4) output:				4	5	□
skip(2) output:			3	4	5	□
ignore() output:						□
ignore(3) output:			□			
search(x -> x ≥ 2) output:		2 □				

The query takeUntil(p) : Q(A, A), where p is a predicate over A, computes like the identity transformation while there is no occurrence of an item satisfying p in the input. When it encounters the first item satisfying p, it emits it to the output and halts. A similar query is take(n) : Q(A, A), where $n \geq 1$ is an integer, which echoes the first n items of the input stream to the output and then halts. The query skipUntil(p) : Q(A, A), for a predicate p on A, emits no output while the input contains no item satisfying p. When the first item satisfying p is seen, it emits it to the output and continues to compute like the identity transformation. The query skip(n) : Q(A, A), for an integer $n \geq 1$, emits no output for the first n input items, and then proceeds to echo the rest of the input stream. The query ignore() : Q(A, A) emits empty output for all input items and halts for the end-of-stream marker. The query ignore(n) : Q(A, A), for an integer $n \geq 1$, emits no output for the first n input items and then immediately halts. For a predicate p on A, the query search(p) : Q(A, A) emits no output while it searches for the first occurrence of an item satisfying p. When it encounters such an item, it emits it to the output and halts. See Table 3.6.

The ***temporal sequencing*** combinator can apply different queries in sequence (i.e., one after the other), thus varying the computation over time. For queries f and g of type Q(A, B), their temporal sequencing seq(f, g) : Q(A, B) computes like f until it

Table 3.7 : Temporal sequencing.

input:	1	2	3		4	3	2		1	□
`f` output:			3	□						
`seq`(`f`, `g`) output:			3		4	3	2	□		

Table 3.8 : Temporal iteration.

input:	2	3	0		9	0	1	7	0	3
`f` output:	2	3	0	□						
`f` ≫ `g` output:			5	□						
`iter`(`f` ≫ `g`) output:			5			9			8	

halts, and then it proceeds to compute like `g`. For example, if `f` = `search`(`x -> x` ≥ 3) and `g` = `takeUntil`(`x -> x` ≤ 2), then `seq`(`f`, `g`) computes as shown in Table 3.7.

The ***temporal iteration*** combinator can be used to repeat a streaming computation indefinitely. For a query `f` : `Q`(A, B), its temporal iteration `iter`(`f`) : `Q`(A, B) executes `f` and restarts it every time it halts. This results in an unbounded temporal repetition of the computation that `f` specifies. Now, the iteration of `f` ≫ `g`, where `f` = `takeUntil`(`x -> x` = 0) and `g` = `reduce`$(0, +)$, computes as shown in Table 3.8.

Flatten and Emit

The query `flatten` : `Q`(`List`$(A), A$) processes an input stream whose data items are lists that contain elements of type A, and it propagates elements in the list to the output. For a list `out` : `List`(B), the query `emit`(`out`) : `Q`(A, B), specifies the computation that outputs the elements of `out` at the very beginning (before any input items are consumed) and then immediately halts. See Table 3.9 for examples.

Table 3.9 : Flatten and Emit.

	input:	$[a_1, a_2]$	[]	$[a_3]$	[]	□
`flatten`	output:	a_1	a_2	a_3		□

	input:		a_1	a_2	...	
`emit`$([b_1, b_2])$	output:	b_1	b_2	□		

Table 3.10 : Zip and ZipLast.

input:	2.4	a	b	3.5	c	-0.9	□
val:	2.4			3.5		-0.9	
id:		a	b		c		
`f` output:		$\langle 2.4, a\rangle$		$\langle 3.5, b\rangle$		$\langle -0.9, c\rangle$	□
`g` output:		$\langle 2.4, a\rangle$	$\langle 2.4, b\rangle$	$\langle 3.5, b\rangle$	$\langle 3.5, c\rangle$	$\langle -0.9, c\rangle$	□

The Join Construct

StreamQL provides the constructs `zip`, `zipLast`, and `join` to combine input data from several input sub-streams. Let us consider an input stream with data items from two different sources, in which one is the signal measurement (of type `V`), and the other is the signal identifier (of type `ID`). The input type is `Or`(`V`, `ID`), which means that an input item is either of type `V` or of type `ID`. Then, to annotate the signal measurements with corresponding identifiers as outputs of type `IV` = {`id` : `ID`, `val` : `V`}, a natural way is to combine the values and the identifiers based on their order of arrival. StreamQL provides the constructs `zip` and `zipLast`. Given a function `op` = (`val`, `id`) `->` $\langle$`val`, `id`$\rangle$: `V` × `ID` → `IV` that annotates a signal measurement with an identifier, the query `f` = `zip`(`op`) : `Q`(`Or`(`V`, `ID`), `IV`) combines the measurements and the identifiers one by one with respect to their order of arrival, and the query `g` = `zipLast`(`op`) : `Q`(`Or`(`V`, `ID`), `IV`) combines the last arrived data items from different categories (See Table 3.10).

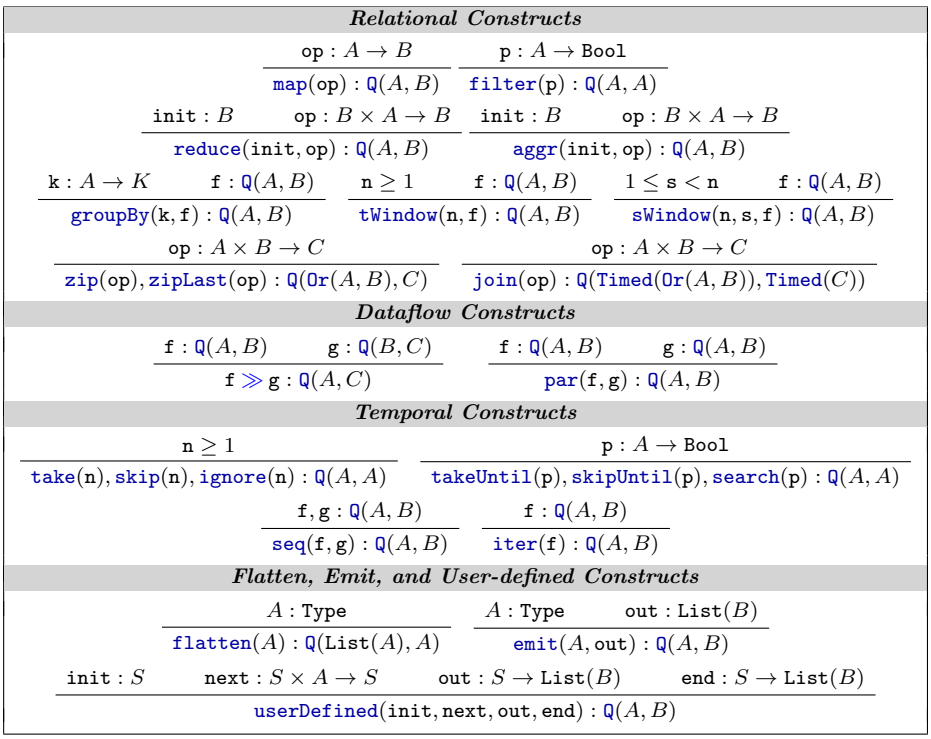


Figure 3.2 : Constructs in StreamQL.

Moreover, StreamQL allows users to assign a *validity interval* to the data item, and it provides the `join` construct to combine data items that have overlapping validity intervals. To assign validity intervals, users need to provide the start/end time of the interval for each input – the input type is specified as $\texttt{Timed}(D) = \{\texttt{data} : D, \texttt{startT} : T, \texttt{endT} : T\}$, where D denotes the type of the data, and T is the type of the time unit (e.g., long integers). Suppose $D = \texttt{Or}(A, B)$ (i.e., the input data is either of type A or type B), given a binary function $\texttt{op} : A \times B \to C$ that combines data, the $\texttt{join}(\texttt{op}) : \texttt{Q}(\texttt{Timed}(\texttt{Or}(A, B)), \texttt{Timed}(C))$ query joins data items with overlapping validity intervals. The output item, of type $\texttt{Timed}(C)$, is also labeled by a validity interval which is the intersection of the validity intervals of the input data.

User-defined Construct

The construct `userDefined` is used to specify a stream transformation with a transducer (state machine). The query $\texttt{userDefined}(\texttt{init}, \texttt{next}, \texttt{out}, \texttt{end}) : \texttt{Q}(A, B)$ takes four arguments to describe the computation: `init` (of type S) is the initial state of the transducer, $\texttt{next} : S \times A \to S$ is the state transition function, $\texttt{out} : S \to \texttt{List}(B)$ is the output function, and $\texttt{end} : S \to \texttt{List}(B)$ gives the final output (upon the termination of the input with $\square$).

Program ECG Peak Detection in StreamQL

Figure 3.2 summarizes several constructs of StreamQL. At the beginning of this section, we gave a high-level description of a simple streaming algorithm for detecting the peaks in the ECG signal. We will now use StreamQL to provide a complete description of this algorithm, which is a variant of the SQRS algorithm [209]. Later in Section 3.8, we will present a significant example for processing the Arterial Blood Pressure signal. Suppose that the data stream concerns multiple patients, that is, it is the interleaving of several ECG time series, one for each patient. The type of the input data items is a record type $\texttt{IVT} = \{\texttt{id} : \texttt{ID}, \texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}\}$, where `ID` is the type of patient identifiers, `V` is the type of scalar values, and `T` is the type of time points. At the top level, the algorithm partitions the input stream into several sub-streams, one for each patient, and performs peak detection for each one of these sub-streams independently. The query $\texttt{groupBy}(\texttt{x -> x.id}, \texttt{findPeak})$ describes this computation, where `findPeak` specifies the peak detection algorithm for a single-patient ECG data stream. This is defined as $\texttt{findPeak} = \texttt{smooth} \gg \texttt{deriv} \gg \texttt{detect}$, which is the composition of three stages: (1) smoothing the signal, (2) computing derivatives, and (3) detecting peaks. The smoothing query $\texttt{smooth} : \texttt{Q}(\texttt{IVT}, \texttt{IVTF})$ has output type `IVTF`, which is the record

type IVT extended with the component fval : V for storing the smoothed (low-pass filtered) value.

$$\texttt{smooth} = \texttt{sWindow}(5, 1, (\texttt{v}, \texttt{w}, \texttt{x}, \texttt{y}, \texttt{z})\ \texttt{->}\ \texttt{expr}),\ \text{where}$$

$$\texttt{expr} = \langle \texttt{x.id}, \texttt{x.val}, \texttt{x.ts}, \texttt{fval} \rangle : \texttt{IVTF}\ \text{and}$$

$$\texttt{fval} = (\texttt{v.val} + 2 \cdot \texttt{w.val} + 4 \cdot \texttt{x.val} + 2 \cdot \texttt{y.val} + \texttt{z.val})/10.$$

The idea is that for a sample x at time x.ts we consider the window (v, w, x, y, z) centered around x and calculate a weighted average over the window for the smoothed value. The differentiation query deriv : Q(IVTF, IVTFD) calculates discrete derivatives by taking the difference of successive smoothed values. It is implemented as follows:

$$\texttt{deriv} = \texttt{sWindow}(2, 1, (\texttt{x}, \texttt{y})\ \texttt{->}\ \texttt{expr}),\ \text{where}$$

$$\texttt{expr} = \langle \texttt{y.id}, \texttt{y.val}, \texttt{y.ts}, \texttt{y.fval}, \texttt{dval} \rangle : \texttt{IVTFD}\ \text{and}\ \texttt{dval} = \texttt{y.fval} - \texttt{x.fval} : \texttt{V}.$$

The record type IVTFD extends IVTF with dval : V for storing the derivative. The detection of the first peak involves searching for the first time point ℓ_1 when dval exceeds the threshold hTh. The signal interval from this point until the time point r_1 when dval falls below the threshold lTh contains the first peak. Thus, the signal in the interval $[\ell_1, r_1]$ is streamed to the argmax query (see below), which finds the data item with the highest value (in the raw, unfiltered signal). This process is repeated indefinitely in order to detect all peaks:

$$\texttt{start} = \texttt{search}(\texttt{x}\ \texttt{->}\ \texttt{x.dval} > \texttt{hTh})$$

$$\texttt{take} = \texttt{takeUntil}(\texttt{x}\ \texttt{->}\ \texttt{x.dval} < \texttt{lTh})$$

$$\texttt{argmax} = \texttt{reduce}((\texttt{x}, \texttt{y})\ \texttt{->}\ (\texttt{y.val} > \texttt{x.val})\ ?\ \texttt{y} : \texttt{x})$$

$$\texttt{detect} = \texttt{iter}(\texttt{seq}(\texttt{start}, \texttt{take}) \gg \texttt{argmax})$$

All four queries above are of type Q(IVTFD, IVTFD).

3.4 Discussion of the Expressiveness of StreamQL

A natural approach for processing a data stream is to write a program in a low-level imperative programming language such as C. However, this process is tedious and error-prone because the computation cannot be easily expressed in a modular way. The program that specifies the computation typically contains complex state-manipulating logic and the code is heavily entangled. For this reason, several domain-specific languages have been proposed which offer various primitive streaming constructs (e.g., pipelines and sliding windows) in order to assist the programmer in expressing the desired computation. In this section, we will illustrate some of the features of StreamQL that facilitate the modular description of streaming computations, particularly for time-series workloads. We will compare StreamQL to both low-level imperative languages (such as C) and domain-specific languages (such as Rx) in the context of a concrete example.

Assume that the input stream consists of signal measurements of type `V` (integer type) which are collected at a fixed frequency. We will consider a computation that is the composition of a smoothing filter and calculating the derivative. We use a low-pass filter to smooth the input into results $\texttt{f} : \texttt{F}$ (floating point type), where $\texttt{f} = (\texttt{v}_1 + 2\texttt{v}_2 + 4\texttt{v}_3 + 2\texttt{v}_4 + \texttt{v}_5)/10$ for each five consecutive input items $\texttt{v}_1, \texttt{v}_2, ..., \texttt{v}_5$. Then, we compute the derivative $\texttt{d} : \texttt{D}$ (floating point type) where $\texttt{d} = \texttt{f}_2 - \texttt{f}_1$ for every two consecutive smoothed values. The top-left part of Figure 3.3 shows the algorithm implemented in C. It processes the input stream item by item by calling the `next` function and produces output items by calling the `out` function. We use the circular array `t` to buffer the input for smoothing. We update the array by replacing its oldest element by the incoming input item, and then we apply the coefficients of the low-pass filter (stored in the `coef` array) to the buffered elements to compute the

```
double coef[5] = {0.1, 0.2, 0.4, 0.2, 0.1};
V t[5]; // circular array
int cnt = 0, start = 0;
bool isFReady = false;
void next(V v){
  // smooth the input
  if (cnt < 5) {
    t[cnt++] = v;
  } else {
    t[start] = v;
    start = (start + 1) % 5;
  }
  F f = 0.0;
  if (cnt == 5) {
    // compute the result when t is full
    for (int i = 0; i < 5; i++) {
      f += coef[i] * t[(start + i) % 5];
    }
  }
  if (isFReady) {
    // compute the derivative
    D d = f - lastF;
    lastF = f;
    out(d); // produce output
  } else if (cnt == 5) {
    lastF = f;
    isFReady = true;
  } // else do nothing
}
```

```
double coef[] = {0.1, 0.2, 0.4, 0.2, 0.1};
class FCnt{
  F f; // FIR filtering result
  int cnt;
  FvalCnt(F f, int cnt){
    this.f = f;
    this.cnt = cnt;
  }
}
Observable<D> outputStream = inputStream
  .window(5, 1) // smooth the input
  .flatMap(wnd -> wnd.reduce(
    new FCnt(0.0, 0),
    (pair, v) -> {
      F f = pair.f;
      int cnt = pair.cnt;
      f += v * coef[cnt];
      cnt ++;
      return new FCnt(f, cnt);
    }).map(p -> p.f).toObservable()
  ).window(2, 1) // compute the derivative
  .flatMap(wnd -> wnd.reduce(
    new ArrayList<>(),
    (l, f) -> { // add f into list l
      return List.copyOf(l.add(f));
    }).map(l -> l.size() == 2 ?
                l.get(1) - l.get(0) : null)
    .filter(d -> d != null).toObservable()
  );
```

```
Q<V,F> smooth = sWindow(5, 1, (a, b, c, d, e) -> (a + 2*y + 4*c + 2*d + e) / 10.0);
Q<F,D> deriv = sWindow(2, 1, (a, b) -> b - a);
Q<V,D> query = pipeline(smooth, deriv);
```

Figure 3.3 : Program for input preprocessing written in C (top-left), RxJava (top-right), and StreamQL (bottom).

smoothing results. After that, the program computes the derivatives and produces the output. The top-right part of Figure 3.3 shows the RxJava implementation. RxJava does not provide a sliding window construct that uses circular arrays. To integrate this efficient data structure, a user of the library would need to create a customized operator from scratch. The bottom part of Figure 3.3 shows the implementation in StreamQL, where the `sWindow` construct allows users to directly aggregate all elements inside the window with efficient built-in data structures.

Now, let us consider an algorithm for detect peaks in a stream of numerical values (suppose they are of type `V`). The algorithm searches for the first value that exceeds

```
V peak = -INFINITY;
enum mode { beforePeak, inPeak, afterPeak };
int cnt;
enum mode m = beforePeak;
void next(V v){
  if (m == beforePeak) {
    if (v > THRESH) {
      m = inPeak;
      cnt = PEAK_CNT;
    } // else do nothing
  } else if (m == inPeak) {
    peak = (v > peak) ? v : peak;
    cnt --;
    if (cnt == 0) {
      out(peak); // produce outputs
      m = afterPeak;
      cnt = SILENCE_CNT;
    }
  } else { // m == afterPeak
    cnt --;
    if (cnt == 0) {
      m = beforePeak;
      peak = -INFINITY;
    }
  }
}
```

```
Q<V,V> start = search(v -> v > THRESH);
Q<V,V> take = take(PEAK_CNT);
Q<V,V> max = reduce((x, y) -> (y > x) ? y : x);
Q<V,V> find1 = pipeline(seq(start, take), max);
Q<V,V> silence = ignore(SILENCE_CNT);
Q<V,V> query = iterate(seq(find1, silence));
```

```
enum Mode { beforePeak, inPeak, afterPeak }
class State{
  Mode mode;            int cnt;
  boolean sendOut;      V peak;
  State(Mode m, int c, boolean s, V p) {
    mode = m;           cnt = c;
    sentOut = s;        peak = p;
  }
}
Observable<V> outputStream = derivStream.scan(
  new State(beforePeak, 0, false, -INFINITY),
  (s, v) -> {
    Mode m = s.mode;
    int cnt = s.cnt;
    boolean sendOut = false;
    V peak = s.peak;
    if (m == beforePeak) {
      if (v > THRESH) {
        m = inPeak;
        cnt = PEAK_CNT;
      } // else do nothing
    } else if (m == inPeak) {
      peak = (v > peak) ? v : peak;
      if (-- cnt == 0) {
        sendOut = true;
        m = afterPeak;
        cnt = SILENCE_CNT;
      }
    } else { // m == afterPeak
      if (-- cnt == 0) {
        m = beforePeak;
        peak = -INFINITY;
      }
    }
    return new State(m, cnt, sendOut, peak);
  }).filter(s -> s.sendOut).map(s -> s.peak);
```

Figure 3.4 : Program for peak detection written in C (top-left), RxJava (right), and StreamQL (bottom-left).

the threshold `THRESH`. Then, it search for the maximum over the next #`PEAK_CNT` elements, which is considered a peak. After that, the algorithm silences detection for #`SILENCE_CNT` elements to avoid a duplicate detection. This process is repeated indefinitely in order to detect all peaks. The top-left part of Figure 3.4 shows the C implementation of the algorithm, where the input stream is repeatedly partitioned into three regions: `beforePeak`, `inPeak`, and `afterPeak`. This partitioning is data-dependent as the end of `beforePeak` happens when the value exceeds the threshold. The right part of Figure 3.4 is the RxJava implementation. Rx provides count/time-based tumbling windows to split up the stream into non-overlapping regions. However, such a decomposition is not data-dependent, as it does not rely on the input values. Moreover, Rx has no operator like StreamQL's `iter` for repeating the execution of a query (i.e., detection of a single peak) every time it halts. Given the absence of these features, the most convenient way to program the algorithm in RxJava is to use its `scan` operator (similar to `aggr` in StreamQL). This amounts to providing a monolithic imperative implementation of the whole algorithm. The bottom-left part of Figure 3.4 shows the implementation of the algorithm in StreamQL.

Semantically, Rx and StreamQL can be viewed as algebras with combinators. There is a key difference. Rx is an algebra of streams, where the basic objects are streams and the combinators are operations on streams. StreamQL, on the other hand, is an algebra of stream transformations, where its basic objects are stream transformations and the combinators are operations on transformations. More specifically, `Observable` is the basic object in Rx that represents a stream. Rx describes the overall computation as a sequence of transformations to the source `Observable`. The first-class object in StreamQL is the stream transformation, which is captured syntactically with a query. StreamQL describes the computation as the composition of sub-computations defined

	StreamQL	Rx	Siddhi	Trill
1. stream filtering	yes	yes	yes	yes
2. stream mapping	yes	yes	yes	yes
3. sequential aggregation	yes	yes	yes	yes
4. key-based partitioning	yes	yes	yes	yes
5. tumbling window	yes	yes	yes	yes
6. sliding window	yes	yes	yes	yes
7. efficient window aggregation	yes	no	yes	yes
8. streaming pipeline	yes	yes	yes	yes
9. relational join	yes	yes	yes	yes
10. temporal sequencing	yes	no	no	no
11. temporal iteration	yes	no	no	no
12. signal processing primitives	yes	no	no	yes
13. regular parsing	yes	no	yes	yes
14. user-defined functions	yes	yes	yes	yes

Figure 3.5 : Some of the features and streaming constructs supported by StreamQL, Rx, Siddhi, and Trill.

by queries. For example, `iter(f)` is the temporal iteration of a query `f`. So, if Rx is considered first-order then StreamQL is second-order. This explains why `iter` and `seq` are easily integrated into StreamQL but are more difficult to express in Rx.

Figure 3.5 lists some useful constructs for stream processing and the engines that support them. The streaming operations marked with $*$ in Figure 3.5 indicate that the library supports such operations, but their use requires additional encoding. For example, for *sequential aggregation*, Siddhi does not have a construct like StreamQL's `aggr`. Instead, it defines a set of fixed aggregations (e.g., sum and average). Other sequential aggregations can be implemented using user-defined functions. Rx does not implement efficient algorithms for *window aggregation*, which is discussed in detail in Section 3.7. The *temporal sequencing* and the *temporal iteration* constructs (`seq` and `iter` in StreamQL) decompose the stream into sub-streams, and the decomposition is data-dependent. It is not easy to express such computations in Rx, Siddhi, and Trill

in a modular way. Both StreamQL and Trill provide *signal processing* constructs (e.g., FFT, FIR, and IIR filtering) to transform and analyze signals.

3.5 Denotational Semantics of StreamQL

In this section, we present the denotational semantics of StreamQL using a class of monotone functions. This semantics clarifies the meaning of the language primitives and combinators. The use of monotone functions or other sequence transductions for describing streaming computations has been considered in [200, 210, 211, 38] and in a much more general algebraic setting in [212].

3.5.1 Formalization of Data Stream and Stream Transformation

A data stream can be viewed as a potentially unbounded sequence of data items that may or may not terminate. Distinguishing the termination of a data stream proves useful in real-world scenarios, as streams often eventually come to an end. For instance, when a processing system logs in a streaming fashion, the data stream terminates if there are no more entries in the input file. For a type A, we write A^* to denote the set of finite sequences over A. We write $u \cdot v$ or uv to denote the concatenation of the sequences u and v, and ε for the empty word. We use the special symbol $\Box$ to indicate the end of a stream, which is called the *end-of-stream marker*. We define $A^\dagger = A^* \cdot \{\varepsilon, \Box\} = A^* \cup (A^* \cdot \Box)$ as the type of a data stream, i.e., $A^\dagger$ contains the finite sequences over A that could potentially end with an end-of-stream marker. For sequences $x, y \in A^\dagger$, we write $x \leq y$ if x is a prefix of y, i.e. $xz = y$ for some $z \in A^\dagger$. We say that $\leq$ is the *prefix relation* on sequences. When $x \leq y$, there is a *unique* z with $xz = y$, which we denote by $x^{-1}y$. We write $x < y$ when $x \leq y$ and $x \neq y$. A sequence $x \in A^\dagger$ is said to be *terminated* if it ends with $\Box$.

Given $A^\dagger$ as the type of a data stream, the input/output behavior of a streaming computation can be described semantically by a function of type $A^\dagger \to B^\dagger$, where A is the type of the input items and B is the type of the output items. If $x \in A^\dagger$ is the prefix of the input stream seen so far (which we also call the cumulative input), then $f(x) \in B^\dagger$ is the cumulative output that has been emitted after the whole sequence x is processed. As more input data items arrive, the output stream gets extended with more output items. This is captured formally by requiring that the function f is *monotone*: $x \leq y$ implies that $f(x) \leq f(y)$ for every $x, y \in A^\dagger$. A monotone function $f : A^\dagger \to B^\dagger$ is said to be a ***stream transformation***. We write $\mathsf{ST}(A, B)$ to denote the set of all stream transformations with input (resp., output) data items of type A (resp., B).

As mentioned earlier, a stream transformation $f : \mathsf{ST}(A, B)$ specifies the *cumulative output* of a streaming computation, i.e. the total output that has been emitted from the beginning until the entire cumulative input is consumed. The computation can be described equivalently by specifying the *incremental output*, i.e. the output increment that is emitted exactly when the last item of a cumulative input is consumed. The incremental output of f for the cumulative input xa is equal to $f(x)^{-1}f(xa)$.

incremental input	cumulative input	incremental output	cumulative output
	ε	0	0
1	1	1	0 1
2	1 2	3	0 1 3
3	1 2 3	6	0 1 3 6
□	1 2 3 □	□	0 1 3 6 □

The table above illustrates these concepts with the example of calculating the running sum over a stream of integers. Suppose $f : \mathsf{ST}(A, B)$ describes the input/output

behavior of a streaming computation in a cumulative fashion, and $\varphi : A^\dagger \to B^\dagger$ describes the same computation in an incremental fashion. Then, f and φ are related in the following way:

$$f(a_1 a_2 \ldots a_n) = \varphi(\varepsilon) \cdot \varphi(a_1) \cdot \varphi(a_1 a_2) \cdots \varphi(a_1 a_2 \ldots a_n)$$
$$f(a_1 a_2 \ldots a_n \Box) = f(a_1 a_2 \ldots a_n) \cdot \varphi(a_1 a_2 \ldots a_n \Box)$$

for all $a_1 a_2 \ldots a_n \in A^*$. Equivalently, we have that

$$\varphi(\varepsilon) = f(\varepsilon) \qquad \varphi(ua) = f(u)^{-1} f(ua) \qquad \varphi(u\Box) = f(u)^{-1} f(u\Box)$$

for all $u \in A^*$ and $a \in A$. Function φ satisfies the following property: if $\varphi(x)$ ends with $\Box$, then $\varphi(y) = \varepsilon$ for all $y \geq x$. This says that when the output stream terminates, no more output data items can be emitted. We write $\partial f : A^\dagger \to B^\dagger$ to denote the incremental version of $f : \mathsf{ST}(A, B)$.

Figure 3.6 gives the denotational semantics for some core combinators of StreamQL. The definition of the stream transformations $\mathsf{map}(\mathsf{op})$ and $\mathsf{filter}(\mathsf{p})$ are straightforward. The transformations $\mathsf{reduce}(b, \mathsf{op})$ and $\mathsf{aggr}(b, \mathsf{op})$ are both aggregations, but differ in when they give output. Informally, $\mathsf{reduce}(b, \mathsf{op})$ gives the total aggregate when the stream terminates, whereas $\mathsf{aggr}(b, \mathsf{op})$ gives the running aggregate every time a new item arrives. Their definition requires the *fold combinator* $\mathsf{fold} : B \times (B \times A \to B) \times A^* \to B$, given by $\mathsf{fold}(b, \mathsf{op}, \varepsilon) = b$ and $\mathsf{fold}(b, \mathsf{op}, ua) = \mathsf{op}(\mathsf{fold}(b, \mathsf{op}, u), a)$. The *streaming (serial) composition* combinator is given by:

$$\frac{f : \mathsf{ST}(A, B) \qquad g : \mathsf{ST}(B, C)}{f \gg g : \mathsf{ST}(A, C)} \qquad (f \gg g)(a) = g(f(a))$$

$$\frac{\mathtt{op} : A \to B}{\begin{array}{r} f = \mathsf{map}(\mathtt{op}) : \mathsf{ST}(A, B) \\ f(\varepsilon) = \varepsilon \\ f(ua) = f(u) \cdot \mathtt{op}(a) \\ f(u\Box) = f(u)\Box \end{array}} \qquad \frac{\mathtt{p} : A \to \mathtt{Bool}}{\begin{array}{r} f = \mathsf{filter}(\mathtt{p}) : \mathsf{ST}(A, A) \\ f(\varepsilon) = \varepsilon,\ f(u\Box) = f(u)\Box \\ f(ua) = f(u) \cdot a,\ \text{if } \mathtt{p}(a) = \mathtt{true} \\ f(ua) = f(u),\ \text{if } \mathtt{p}(a) = \mathtt{false} \end{array}} \qquad \frac{b : B \qquad \mathtt{op} : B \times A \to B}{\begin{array}{r} f = \mathsf{aggr}(b, \mathtt{op}) : \mathsf{ST}(A, B) \\ f(\varepsilon) = \varepsilon \\ f(ua) = f(u) \cdot \mathsf{fold}(b, \mathtt{op}, ua) \\ f(u\Box) = f(u)\Box \end{array}}$$

$$\frac{b : B \qquad \mathtt{op} : B \times A \to B}{\begin{array}{r} f = \mathsf{reduce}(b, \mathtt{op}) : \mathsf{ST}(A, B) \\ f(u) = \varepsilon \\ f(u\Box) = \mathsf{fold}(b, \mathtt{op}, u)\Box \end{array}} \qquad \frac{A : \mathtt{Type}}{\begin{array}{r} f = \mathsf{flatten}(A) : \mathsf{ST}(\mathsf{List}(A), A) \\ (\partial f)(\varepsilon) = \varepsilon,\ (\partial f)(u\Box) = \Box \\ (\partial f)(ul) = \mathsf{extract}(l) \end{array}}$$

$$\frac{A : \mathtt{Type} \qquad \mathtt{out} : \mathsf{List}(B)}{\begin{array}{r} f = \mathsf{emit}(A, \mathtt{out}) : \mathsf{ST}(A, B) \\ (\partial f)(\varepsilon) = \mathsf{extract}(\mathtt{out})\Box \\ (\partial f)(u) = \varepsilon,\ \text{if } |u| > 0, \\ (\partial f)(u\Box) = \varepsilon \end{array}} \qquad \frac{f : \mathsf{ST}(A, B) \qquad f \uparrow \varepsilon}{\begin{array}{r} g = \mathsf{iter}(f) : \mathsf{ST}(A, B) \\ g(x) = f(x),\ \text{if } f \uparrow x \\ g(ux) = f(u)\Box^{-1} \cdot g(x),\ \text{if } f \Downarrow u \\ g(u\Box) = f(u\Box)\Box^{-1} \cdot f(\varepsilon),\ \text{if } f \Downarrow u\Box \end{array}} \qquad \frac{f : \mathsf{ST}(A, B) \qquad g : \mathsf{ST}(A, B)}{\begin{array}{r} h = \mathsf{seq}(f, g) : \mathsf{ST}(A, B) \\ h(x) = f(x),\ \text{if } f \uparrow x \\ h(ux) = f(u)\Box^{-1} \cdot g(x),\ \text{if } f \Downarrow u \\ h(u\Box) = f(u\Box)\Box^{-1} \cdot g(\varepsilon),\ \text{if } f \Downarrow u\Box \end{array}}$$

$$\frac{\mathtt{p} : A \to \mathtt{Bool}}{\begin{array}{r} f = \mathsf{takeUntil}(\mathtt{p}) : \mathsf{ST}(A, A) \\ (\partial f)(\varepsilon) = \varepsilon \text{ and } (\partial f)(u\Box) = \varepsilon \\ (\partial f)(ua) = \varepsilon,\ \text{if } \mathtt{p}'(u) = \mathtt{true} \\ (\partial f)(ua) = a,\ \text{if } \mathtt{p}'(u) = \mathtt{false} \text{ and } \mathtt{p}(a) = \mathtt{false} \\ (\partial f)(ua) = a\Box,\ \text{if } \mathtt{p}'(u) = \mathtt{false} \text{ and } \mathtt{p}(a) = \mathtt{true} \end{array}} \qquad \frac{f : \mathsf{ST}(A, B) \qquad g : \mathsf{ST}(A, B)}{\begin{array}{r} h = \mathsf{par}(f, g) : \mathsf{ST}(A, B) \\ (\partial h)(u) = (\partial f)(u)\Box^{-1} \cdot (\partial g)(u)\Box^{-1},\ \text{if } f \uparrow u \text{ or } g \uparrow u. \\ (\partial h)(u) = (\partial f)(u)\Box^{-1} \cdot (\partial g)(u)\Box^{-1}\Box,\ \text{if } f \downarrow u \text{ and } g \Downarrow u. \\ (\partial h)(u) = (\partial f)(u)\Box^{-1} \cdot (\partial g)(u)\Box^{-1}\Box,\ \text{if } f \Downarrow u \text{ and } g \downarrow u. \\ (\partial h)(u) = \varepsilon,\ \text{otherwise} \end{array}}$$

$$\frac{\mathsf{key} : A \to K \qquad f : \mathsf{ST}(A, B)}{\begin{array}{r} g = \mathsf{groupBy}(\mathsf{key}, f) : \mathsf{ST}(A, B) \\ (\partial g)(\varepsilon) = \varepsilon \\ (\partial g)(ua) = (\partial f)(u|_{\mathsf{key}(a)} \cdot a)\Box^{-1} \\ (\partial g)(u\Box) = \left(\prod_{i=1}^{n} (\partial f)(u|_{k_i}\Box)\Box^{-1}\right)\Box \end{array}} \qquad \frac{n \geq 1 \qquad f : \mathsf{ST}(A, B)}{\begin{array}{r} g = \mathsf{tWindow}(n, f) : \mathsf{ST}(A, B) \\ g(u) = f(u)\Box^{-1},\ \text{if } |u| < n \\ g(u\Box) = g(u)\Box,\ \text{if } |u| < n \\ g(ux) = f(u\Box)\Box^{-1} \cdot g(x),\ \text{if } |u| = n \end{array}}$$

Figure 3.6 : Semantics of map, filter, reduce, aggr, seq, iter, emit, flatten, takeUntil, par, groupBy and tWindow.

We write $\gg$ to denote the composition of functions. For a stream transformation $f : \mathsf{ST}(A, B)$, we write $f \downarrow x$ to indicate that $f(x)$ is terminated, and $f \uparrow x$ to mean that $f(x)$ is not terminated. We say that f *halts* on $x \in A^{\dagger}$, denoted $f \Downarrow x$, if the following hold: (1) $f(x)$ is terminated, and (2) $f(y)$ is not terminated for every $y < x$. For a sequence $u \in A^*$, we define $(u\Box) \cdot \Box^{-1} = u$ and $u \cdot \Box^{-1} = u$. In other words, $(- \cdot \Box^{-1})$ is the operation that removes the end-of-stream marker from a sequence if it is present. Using this notation, we define the *temporal sequencing* combinator seq, and the *temporal iteration* combinator iter in Figure 3.6. Notice that $\mathsf{iter}(f)$ is defined under the assumption that $f(\varepsilon)$ is not terminated. This is required, because otherwise

the computation of $\mathsf{iter}(f)$ would enter an infinite loop of halting and restarting without consuming any input. The definition of $\mathsf{groupBy}(\mathsf{key}, f)$ in Figure 3.6 uses the incremental viewpoint for notational brevity. For a sequence $u \in A^*$ and a key $k \in K$, we write $u|_k$ to denote the subsequence of u that contains the items whose key is equal to k. More formally, $\varepsilon|_k = \varepsilon$, $(ua)|_k = u|_k$ if $\mathsf{key}(a) \neq k$, and $(ua)|_k = u|_k \cdot a$ if $\mathsf{key}(a) = k$. In the third case $(\partial g)(u\Box)$ of the $\mathsf{groupBy}(\mathsf{key}, f)$ definition, we use $\prod$ as a generalization of concatenation to arbitrarily many arguments. Moreover, $k_1, k_2, \ldots, k_n$ is taken to be the sequence of keys that appear in the sequence u (in their order of appearance). The definition of the transformations emit and $\mathsf{flatten}$ both require the *extract combinator* $\mathsf{extract} : \mathsf{List}(A) \to A^*$, given by $\mathsf{extract}(\mathsf{nil}) = \varepsilon$ and $\mathsf{extract}(\mathsf{cons}(a, l)) = a \cdot \mathsf{extract}(l)$. In the definition of $\mathsf{takeUntil}(\mathtt{p})$, we lift the predicate $\mathtt{p}$ on A to the predicate $\mathtt{p}'$ on A^*, where, for a sequence $u \in A^*$, $\mathtt{p}'(u) = \mathtt{true}$ if there exists an item a in u such that $\mathtt{p}(a) = \mathtt{true}$, and $\mathtt{p}'(u) = \mathtt{false}$ if for all a in u such that $\mathtt{p}(a) = \mathtt{false}$. In the definition of the parallel composition, given stream transformations f and g, $\mathsf{par}(f, g)$ emits the end-of-stream marker only when f and g have both terminated. Finally, in the definition of $\mathsf{tWindow}(n, f)$ and $\mathsf{emit}(\mathsf{out})$ in Figure 3.6, $|u|$ denotes the length of u.

Theorem 3.5.1 (**Expressive Completeness).** Let $f : \mathsf{ST}(A, B)$ be a stream transformation. If f is computable, then there is a query of type $\mathtt{Q}(A, B)$ that computes it.

Proof. A streaming algorithm for f can be viewed as an automaton $\mathcal{A} = (S, \mathit{init}, \mathit{next}, \mathit{out})$, where S is a (potentially infinite) state space, $\mathit{init} \in S$ is the initial state, $\mathit{next} : S \times (A \cup \{\Box\}) \to S$ is the state transition function, and $\mathit{out} : S \to B^\dagger$ is the output function. We put $S = A^\dagger$, $\mathit{init} = \varepsilon$, $\mathit{next}(s, a) = sa$, $\mathit{next}(s, \Box) = s\Box$, $\mathit{out}(s) = (\partial f)(s)$, and $\mathit{out}(s\Box) = (\partial f)(s\Box)$ for every $s \in A^*$ and $a \in A$. The execution of $\mathcal{A}$ is an

obvious generalization of the execution of finite-state automata. Since f is computable, so are *next* and *out*. It remains to show that the execution of $\mathcal{A}$ can be encoded by a query of type $\mathtt{Q}(A, B)$. Let $\delta : S \times A \to S$ be the restriction of *next* to $S \times A$. Define

$$\mathtt{f} = \mathtt{par}(\mathtt{emit}(A, [init]), \mathtt{aggr}(init, \delta), \mathtt{reduce}(init, \delta) \gg \mathtt{map}(x \texttt{->} next(x, \square)) : \mathtt{Q}(A, S).$$

The query `f` transforms the stream of input items (of type A) into the stream of states (of type S) that the automaton $\mathcal{A}$ goes through. Let $\vartheta : S \to \mathtt{Bool}$ be the function that indicates whether a state is halting or not, that is, for all $s \in S$, $\vartheta(s) = \mathtt{true}$ iff $out(s)$ ends with $\square$. Then, the query $\mathtt{g} = \mathtt{takeUntil}(\vartheta) : \mathtt{Q}(S, S)$ takes a stream of states and echoes them up until (and including) the first halting state. Let $o : S \to B^*$ be given by $o(s) = out(s) \cdot \square^{-1}$. The query $\mathtt{h} = \mathtt{flatten}(\mathtt{map}(o)) : \mathtt{Q}(S, B)$ takes a stream of states as inputs and emits the corresponding flattened output. Finally, the query $\mathtt{f} \gg \mathtt{g} \gg \mathtt{h} : \mathtt{Q}(A, B)$ computes the stream transformation f. $\square$

We will use an example to illustrate the construction in the proof of Theorem 3.5.1. Suppose the input is $\bar{a} = a_1 a_2 a_3 \square$. Define the states $s_0 = init$, $s_{i+1} = next(s_i, a_{i+1})$, and $t_i = next(s_i, \square)$. The output of `f` on $\bar{a}$ is $\bar{s} = s_0 s_1 s_2 s_3 t_3 \square$. Suppose that s_2 is the first halting state. Then, the output of `g` for input $\bar{s}$ is $\bar{t} = s_0 s_1 s_2 \square$. Finally, the output of `h` for input $\bar{t}$ is $o(s_0) \cdot o(s_1) \cdot o(s_2) \cdot \square = f(\bar{a})$.

3.6 Implementation of StreamQL

In this section, we will describe the implementation of StreamQL. We have chosen to implement StreamQL as an embedded domain-specific language in Java (effectively a Java library) in order to provide easy integration with user-defined types and operations. The implementation covers all the core constructs we introduced in Section 3.3 and

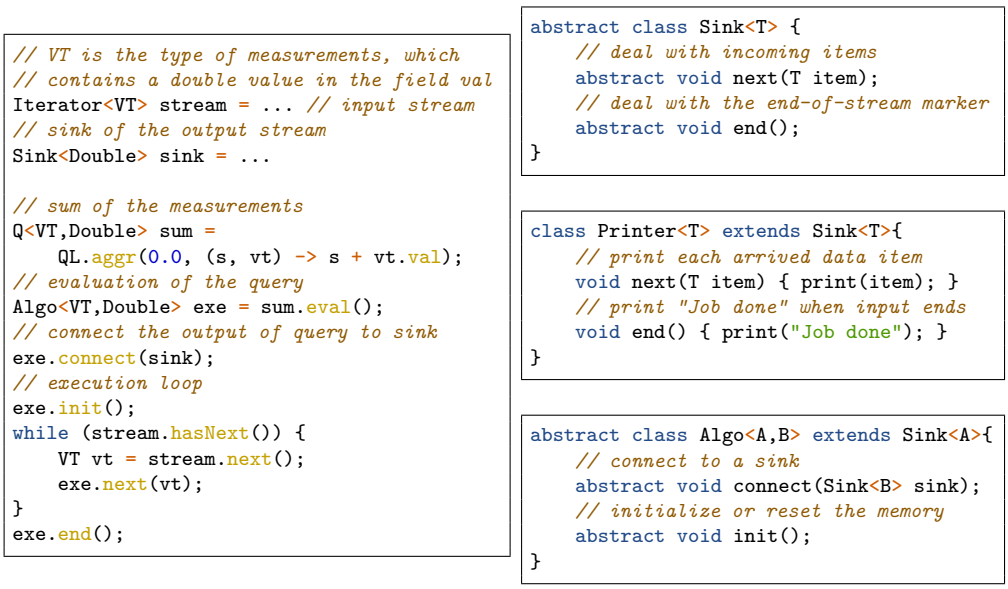
also provides a rich set of specialized algorithms for real-world applications, such as efficient algorithms for aggregation over windows and a variety of signal processing primitives: FFT (Fast Fourier Transform), Hilbert Transform, FIR (Finite Impulse Response) filters, and IIR (Infinite Impulse Response) filters.

The left part of Figure 3.7 gives a simple example of a StreamQL program in Java. Given a signal measurement of type `VT` that contains a double value in the field of `val`, the query `sum` of type `Q` computes the sum of the values of the measurements. The method `eval` returns an object that encapsulates the evaluation algorithm for the query. The methods `init` and `next` are used to initialize the memory and consume data items. When the input stream terminates, the `end` method is invoked.

We define two interfaces, `Sink` and `Algo`, to describe the streaming computation in a push-based manner. The `Algo` interface is used to implement stream transformations. The `Sink` interface is similar to the `Observer` interface of Rx. It is used for specifying a sink that consumes a stream. A sink consumes a stream with two methods, `next` and `end`, that are used for stream elements and the end-of-stream marker respectively. The top-right part of Figure 3.7 shows the definition of the `Sink` interface in Java, and the mid-right part presents an instance of `Sink` that prints each incoming data item and the end-of-stream marker to the console.

The `Algo` interface is used for describing the evaluation algorithm of a query. An implementation of `Algo` specifies how the input stream is transformed into the output stream. The bottom-right of Figure 3.7 shows the definition of the `Algo` interface, which extends the `Sink` interface because it consumes a stream. The `connect` method connects the algorithm to a sink, and the `init` method initializes/resets the state of the algorithm.

Libraries like RxJava and Trill use nested streams (e.g., `Observable`⟨`Observable`⟩



```
// VT is the type of measurements, which
// contains a double value in the field val
Iterator<VT> stream = ... // input stream
// sink of the output stream
Sink<Double> sink = ...

// sum of the measurements
Q<VT,Double> sum =
    QL.aggr(0.0, (s, vt) -> s + vt.val);
// evaluation of the query
Algo<VT,Double> exe = sum.eval();
// connect the output of query to sink
exe.connect(sink);
// execution loop
exe.init();
while (stream.hasNext()) {
    VT vt = stream.next();
    exe.next(vt);
}
exe.end();
```

```
abstract class Sink<T> {
    // deal with incoming items
    abstract void next(T item);
    // deal with the end-of-stream marker
    abstract void end();
}
```

```
class Printer<T> extends Sink<T>{
    // print each arrived data item
    void next(T item) { print(item); }
    // print "Job done" when input ends
    void end() { print("Job done"); }
}
```

```
abstract class Algo<A,B> extends Sink<A>{
    // connect to a sink
    abstract void connect(Sink<B> sink);
    // initialize or reset the memory
    abstract void init();
}
```

Figure 3.7 : The left part shows an example that computes the sum of a nonempty sequence of measurements, the top-right part shows the `Sink` interface, the mid-right part shows an instance of `Sink`, and the bottom-right shows the `Algo` interface.

in RxJava) to decompose the input stream into windows or partitioned sub-streams. StreamQL, on the other hand, eliminates the overheads introduced by the construction of nested streams. For example, the left part of Figure 3.8 presents the algorithm for implementing the tumbling window combinator. Given the size of the window and a sub-query, the tumbling window combinator splits the stream into contiguous non-overlapping windows and applies the sub-query to data items in each window. We provide an algorithm for implementing the tumbling window with a small memory footprint. As shown in the left of Figure 3.8, our algorithm sends incoming data items to the sub-query and maintains a counter to count the number of data items in the window. When the window is full, the algorithm resets the internal state of the sub-query. In contrast, to implement a tumbling window, libraries like RxJava and Trill will construct nested streams to decompose the input stream as several stream

```
class TWnd<A,B> extends Algo<A,B> {
  private final int size;
  // algorithm of the sub-query
  private final Algo<A,B> algo;
  private Sink<B> sink;
  // counter of items in the window
  private int cnt;
  TWnd(int size, Algo<A,B> algo) {
    this.size = size;
    this.algo = algo;
  }
  void connect(Sink<B> sink) {
    this.sink = sink;
    // sink for the sub-query that transfers
    // its output to the output of TWnd
    Sink<B> subSink = new Sink<B>() {
      void next(B item) { sink.next(item); }
      // discard the end-of-stream marker
      void end() { }
    };
    algo.connect(subSink);
  }
  void init() {  cnt = 0;  }
  void next(A item) {
    // reset algo if old window is full
    if (cnt == 0) {  algo.init();  }
    algo.next(item);
    cnt = (cnt + 1) % size;
    if (cnt == 0) {  algo.end();  }
  }
  void end() {  sink.end();  }
}
```

```
class Seq<A,B> extends Algo<A,B>{
  // algorithms of the sub-queries
  private final Algo<A,B> left;
  private final Algo<A,B> right;
  // pointer of the currently active algorithm
  private Algo<A,B> active;
  Seq(Algo<A,B> left, Algo<A,B> right) {
    this.left = left;
    this.right = right;
  }
  void connect(Sink<B> sink) {
    // sink for the left algorithm that
    // activates right when left terminates
    Sink<B> leftSink = new Sink<B>() {
      void next(B item) {
        sink.next(item);
      }
      void end() {
        active = right;
        right.init();
      }
    };
    left.connect(leftSink);
    right.connect(sink);
  }
  void init() {
    active = left;
    left.init();
  }
  void next(A item) {  active.next(item);  }
  void end() {  active.end();  }
}
```

Figure 3.8 : The Java implementation of the tumbling window (left) and stream sequencing (right) constructs.

objects. Whenever the current window becomes full, a new window will be created as a stream object. Moreover, to produce the output stream, some additional overhead is introduced to merge (`flatMap` in RxJava) the output sub-streams that are created from the individual windows. The construction of nested streams introduces overheads in terms of throughput and memory usage. In Section 3.7, we experimentally validated these overheads. Our Java library avoids the overheads of nested streams for all other computations that involve stream decomposition, such as sliding windows and key-based partitioning.

The `Algo` interface facilitates the implementation of the constructs `seq` and `iter`. Such constructs are difficult to be encoded in RxJava and Trill because they decompose the input stream in a data-dependent way. Recall that a query `seq`$(\mathtt{f}, \mathtt{g})$ starts executing as the query `f`, and after `f` terminates, it continues executing as the query `g`. This computation thus splits the input stream into two parts. Figure 3.8 shows our implementation of the `seq` construct. Notice that all data items are simply routed to the appropriate algorithm/sink without creating any intermediate objects. In the implementation of the method `connect`, we provide a sink for the algorithm `left`, which will activate the algorithm `right` once it terminates. The implementation of `iter` uses similar ideas.

3.7 Experimental Evaluation

We evaluated the performance of our library using four benchmarks: (1) a micro-benchmark that focuses on basic operators, (2) a benchmark for pattern detection in real-time stock market data, (3) the popular NEXMark benchmark [213], and (4) TAQMark for the analysis of high-frequency market data. We compare our implementation with RxJava, Rx.NET, Reactor, Siddhi, and Trill. These are chosen

because they are all lightweight and high-performance streaming engines that offer rich APIs and have well-maintained implementations. RxJava, Reactor, and Siddhi are implemented in Java, while Rx.NET and Trill are implemented in .NET.

Experimental Setup

The experiments were executed in Ubuntu 16.04 LTS on a desktop computer equipped with an Intel Xeon(R) E3-1241 v3 CPU (4 cores) with 16 GB of memory (DDR3 at 1600 MHz). For Java programs, we used version 1.8.0_181 of the JDK, and we set the maximum heap size at 3.5 GB. For .NET programs, we used the NET Core 3.1.100 SDK with C# 8.0. To test the performance of Trill, we set the batch size to 1000 for its columnar representation (as suggested by its official documentation [214]). All data points in our experiments represent the average of at least five runs, with error bars showing the standard deviation.

3.7.1 Overhead of the Construction of Nested Streams

StreamQL distinguishes itself from Rx-like libraries by evading the performance drag caused by the construction of nested streams. To empirically establish the performance advantages of this approach, we conducted an experiment testing the throughput of queries, particularly those that necessitate the decomposition of the input stream, leading to the construction of nested streams in Rx-like libraries. We synthesized an input stream of timestamped integers, designated as $\{\texttt{ts}, \texttt{val}\}$, patterned as "$\{1,1\},\{2,2\},...,\{n,n\}$", where both `ts` (timestamp) and `val` (data value) are integer types, and n is defined as 100 million. We used the tumbling window construct to segment the input into non-overlapping regions, with computing the sum of integers in each region. The throughput of this `twnd(sum)` query was measured across a range

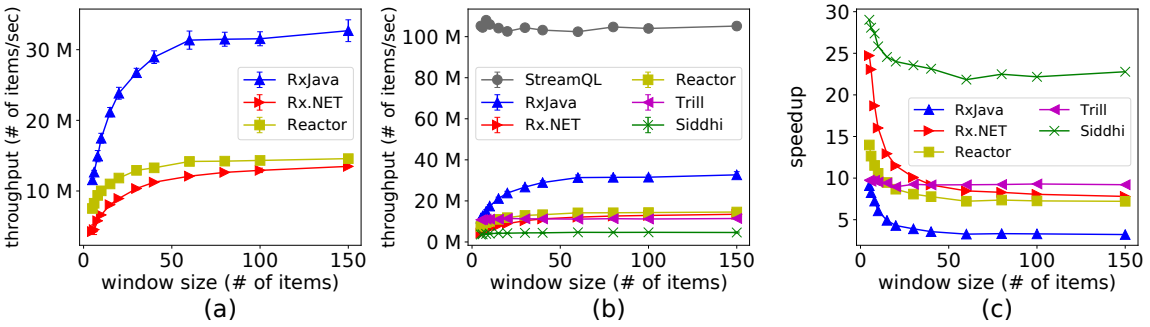


Figure 3.9 : (a) and (b) show the throughput (vertical axis) of `twnd(sum)` queries with different window sizes (horizontal axis), and (c) shows the throughput speedup (vertical axis) of StreamQL compared to other libraries.

of window sizes in our experiments. The window size is the count of integers within a window. Consequently, smaller window sizes result in a higher number of windows, which in turn avoids the generation of nested stream objects in Rx-like libraries.

Figure 3.9 shows the throughput of queries that sum the integers over tumbling windows of various sizes. Figure 3.9(a) presents the throughput in libraries (RxJava, Rx.NET, and Reactor) that decompose the input by nested streams. The results indicate that the construction of nested streams is a significant overhead: when the window size is small (e.g., 4), the throughput is 3 times lower than when the window size is large (e.g., 150). This suggests that the intensive construction of nested streams largely decreases the throughput of stream processing. In Figure 3.9(b), we show the throughput of our StreamQL library along with RxJava, Rx.NET, Reactor, Trill, and Siddhi. In comparison to RxJava, Rx.NET, Reactor, and Siddhi, the throughput of StreamQL queries remains stable with regards to different size of tumbling windows. Finally, Figure 3.9(c) presents the throughput speedup of StreamQL with respect to other libraries. By avoiding the construction of nested streams, StreamQL provides significant performance speedup when the input stream is decomposed into a large

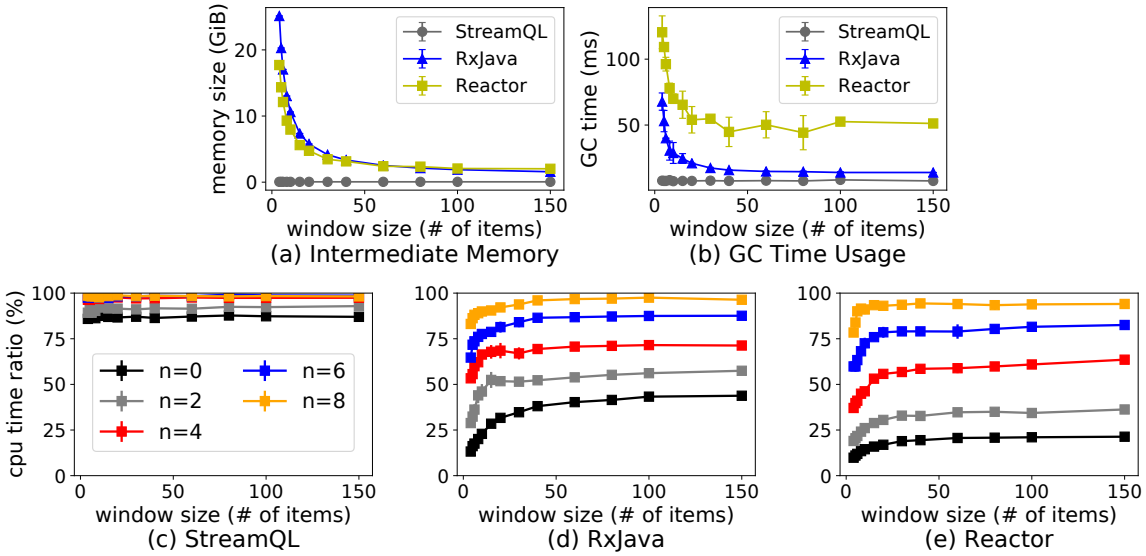


Figure 3.10 : (a) shows the size of intermediate memory (GB, vertical axis) of `twnd(sum)` queries. (b) shows the garbage collection time (ms, vertical axis) of `twnd(sum)` queries. (c), (d), and (e) show the ratio of the execution time on the aggregation calculation to the total execution time for StreamQL, RxJava, and Reactor.

number of windows, and it is more than 3 times faster than Rx-like libraries even when the size of the window is large (e.g. 150) as StreamQL also eliminates the overhead of flattening nested sub-stream objects. The performance of Trill is stable as its windowing operator works by altering the interval timestamp of each stream element. Trill enriches each raw input data item with a "temporal validity" annotation (interval timestamp) to obtain a stream element of type `StreamEvent`. This choice for the temporal and data model has certain semantic advantages, but it also introduces computational costs to incorporate this additional time information.

To further investigate the overheads, we analyzed the memory allocation of the `twnd(sum)` computation for StreamQL, RxJava, and Reactor. We measured the size of the total allocated memory on the heap, and we estimated the memory size for intermediate data structures by subtracting the memory allocated for the input/output

streams from the total allocated memory. Figure 3.10(a) presents the estimation. The StreamQL implementation allocates almost zero additional memory since it uses a counter to record the number of items in the window and maintains the aggregate using a single variable. RxJava and Reactor allocate a significant amount of intermediate memory when the window size is small, which is mainly composed of the nested stream objects (e.g., `InnerObserver` in RxJava), subscription objects (e.g., `UnicastSubject` in RxJava), and internal data buffers (e.g., `SpscLinkedArrayQueue` in RxJava). We also measured the garbage collection (GC) time of the `twnd(sum)` computation. The result is shown in Figure 3.10(b). In addition, we measured the ratio of GC time to total CPU execution time on the main thread, and we observed this ratio is lower than 1% for StreamQL, RxJava, and Reactor. Moreover, we measured the CPU execution time for the `twnd(sum)` query. We estimated the overheads by measuring the time ratio of the cost of the aggregation calculation to the total execution time (higher ratio indicates lower overheads). To illustrate, we aggregated an integer stream using the function $f(agg, x) = agg + 3^n \cdot x$, where n is an integer that controls the complexity of the computation, and when $n = 0$, the aggregation is exactly the sum computation. Figure 3.10(c), (d), and (e) show the results for StreamQL, RxJava, and Reactor. In our observation, when the size of the tumbling window is small (10 items) and the computation per item is cheap ($n = 0$), more than 70% of the time cost in RxJava (80% in Reactor) on the main thread is caused by API calls related with the construction of nested streams, which include the creation of the stream objects (e.g., `InnerObservable.create()` in RxJava), the communication between the stream objects and the corresponding data consumers (e.g., `ObservableFlatMap.drain()` in RxJava), and the management of the data subscriptions (e.g., `Subject.create()` in RxJava).

3.7.2 Efficient Sliding Window Aggregation

The StreamQL library provides efficient algorithms for aggregations over sliding windows. When the aggregation is given by a binary function `op`, then efficient algorithms [215, 216, 217] can be given for the special cases where (1) `op` is associative, and (2) `op` is associative and invertible. Consider the aggregation `max`. The implementation of `max` over a sliding window of size n requires a buffer of size n (to store the contents of the window) [218]. Every time a new item arrives, the naïve algorithm scans through the entire window to calculate the new maximum, which requires $O(n)$ time. Since `max` is associative, there is a better algorithm, which maintains a tree of partial aggregates and only needs $O(\log n)$ time at each step [219]. For a function `op` that is invertible (e.g., sum and count) there is an obvious efficient algorithm, which requires $O(1)$ time at each step ("add" the new item, "subtract" the item falling off the window). Libraries such as Trill and Siddhi also provide efficient algorithms for sliding window aggregation. RxJava, Rx.NET, and Reactor, on the other hand, do not incorporate such algorithms. After creating custom constructs for efficient sliding window aggregation in RxJava, Rx.NET, and Reactor, we compare the performance of our customized constructs with the default constructs. We measure the throughput of queries that sum the integers over sliding windows that have a fixed sliding interval (one item) but different lengths, and we show the results in Figure 3.11. The results suggest that the efficient algorithm is more than 5000 times faster than the default algorithm when the window size is large (e.g., 10,000) and it is about 5 times faster when the size of the window is small.

Remark. To make fair comparisons among StreamQL, Rx, and Reactor, for all queries that involve aggregation over sliding windows in the following benchmarks, we program them using our customized constructs and then test their throughput.

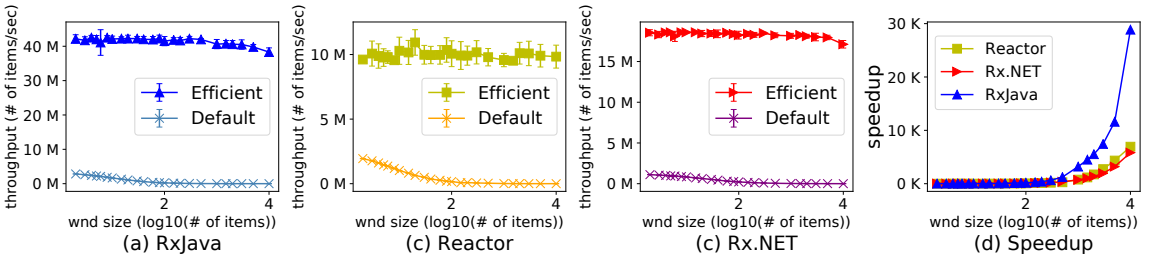


Figure 3.11 : (a), (b), and (c) show the throughput (vertical axis) of `swnd(sum)` queries with different window sizes ($\log_{10}$(# of items), horizontal axis) and a fixed sliding interval in RxJava, Reactor, and Rx.NET. (d) shows the throughput speedup (vertical axis) of efficient implementations compared with the default settings.

3.7.3 Micro Benchmark

We run several basic streaming computations over an input stream of timestamped integers, and the queries are: `map` selects the value of each input item, `filter` removes items with odd integer values, and `sum` calculates the sum of the values. The qualifiers `tw`, `sw`, and `grp` refer to aggregation over tumbling windows, sliding windows, and key-based partitions respectively. The qualifier `gtw`(`gsw`) refers to tumbling (sliding) window aggregation over key-based partitions. All the queries were executed with a stream of timestamped integers. For computations that involve key-based partitioning, we set the key function as $\mathsf{key}(x) = x.\mathtt{val}\ \mathit{mod}\ 100$, and for windows, we always fix the window size to be 100 and the sliding interval to be 1 (if it is a sliding window). Moreover, for sequential aggregation, although StreamQL provides built-in constructs for arithmetical computations, we write queries using primitives to make fair comparisons with libraries that do not provide such features. For example, we use `reduce`$(0, (\mathtt{sum}, \mathtt{x})\ \mathtt{->}\ (\mathtt{sum} + \mathtt{x.val}))$ to compute the sum.

The results are shown in Figure 3.12. (1) StreamQL is 2–100 times faster than Siddhi. The reason for the performance gap is that Siddhi creates complex event

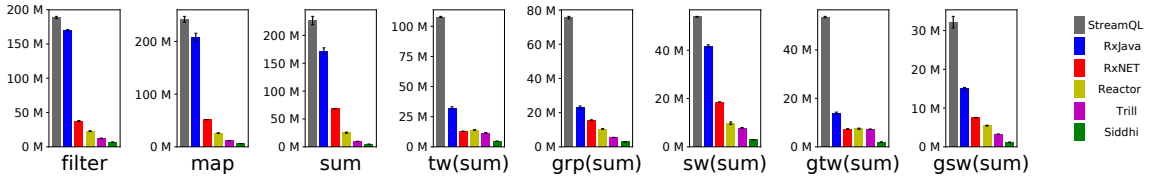


Figure 3.12 : Throughput (# items/sec, vertical axis) of StreamQL, RxJava, Rx.NET, Reactor, Trill, and Siddhi (left to right) in the micro benchmark.

objects to ingress the data and queues the data to achieve streaming composition; both of these bring computational overheads. (2) In comparison to RxJava, for trivial operators (filtering, mapping, and aggregation), there is no significant difference between StreamQL and RxJava. For operations that involve tumbling windows and key-based partitioning, StreamQL is about 2–3 times faster than Rx-like libraries, as it eliminates the overheads brought by nested streams. For sliding window operations, we measured the performance of customized constructs in RxJava in order to make fair comparisons, where the constructs implement efficient algorithms for sliding window aggregation. Therefore, there is no significant difference between StreamQL and RxJava (without these constructs, StreamQL is more than 100 times faster than RxJava). (3) StreamQL is 3–10 times faster than Reactor. In design, Reactor and Rx share many similarities, and Reactor also suffers from the overheads brought by nested streams. (4) In the comparison to Rx.NET and Trill, the results are largely influenced by the performance gap between the Java framework and the .NET framework. Therefore, we can only have a rough comparison between StreamQL and these two libraries.

3.7.4 Stock Benchmark

The stock benchmark [220, 221, 58, 57] uses a synthetic stream of stock quotes that are of the form {`stockId`, `price`, `volume`, `timestamp`}. We consider four families of queries for pattern detection: ***S1*** detects three consecutive quotes whose `volume`s are all above a threshold, ***S2*** detects three consecutive quotes whose `price`s increase continuously, ***S3*** detects five consecutive quotes whose prices fluctuate in a V-pattern (down, down, up, up), and ***S4*** detects price peaks. For every query family there are three variants: ***a.*** concerns a specific stock, ***b.*** detects the pattern for each stock independently, and ***c.*** considers each stock over an 1-minute tumbling window. The experimental results are given in Figure 3.13. For pattern detection that concerns a specific stock (S1a, S2a, S3a, and S4a), StreamQL is about 10-15 times faster than RxJava and Reactor, and 70-100 times faster than Siddhi. When the computation involves excessive stream partitioning (queries labeled by variants b and c), the total computational cost mostly depends on the cost of stream partitioning, and StreamQL is around 3 times faster than RxJava, 4 times faster than Reactor, and 10-20 times faster than Siddhi since it avoids the construction of nested streams.

3.7.5 NEXMark

The NEXMark [213] is about monitoring an on-line auction system. Its data stream has four kinds of events: `Person` represents the registration of a new user, `Item` indicates the start of an auction for a specified item, `Bid` records a bid made for an auctioned item, and `Close` indicates the end of an auction. We used eight queries: ***N1*** converts the price of each bid to another currency, ***N2*** searches for auctions of a specific set of items, ***N3*** counts the number of bids submitted in the US, ***N4*** calculates the average selling price of items for each auction category, ***N5*** outputs the

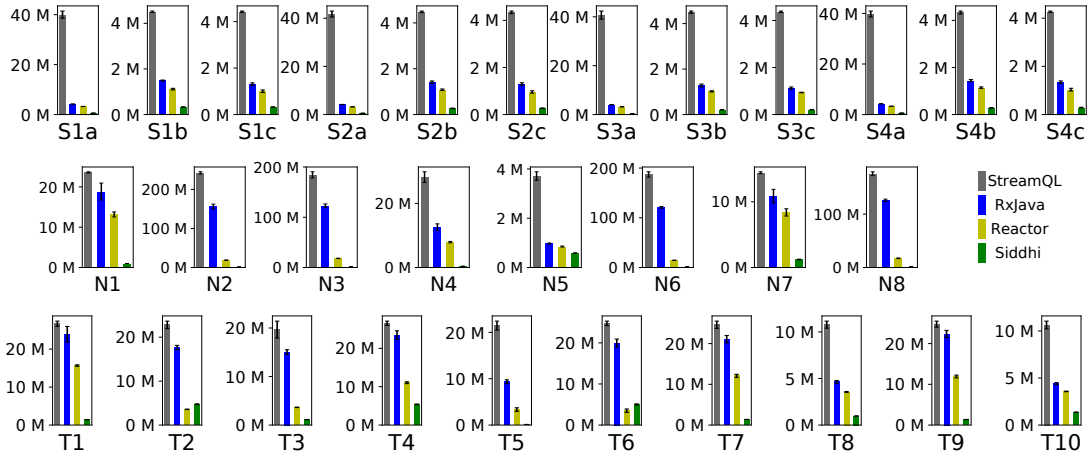


Figure 3.13 : Throughput (# items/sec) of StreamQL, RxJava, Reactor, and Siddhi (left to right) in the stock benchmark (S1a-S4c), NEXMark (N1-N8), and TAQMark (T1-10).

item with the most bids in the last 10 minutes, ***N6*** computes the average selling price per seller for their last 10 closed auctions, ***N7*** finds the highest bid every 1 minute, and ***N8*** calculates, every 12 hours, the number of new user registrations . Figure 3.13 shows the experimental results: StreamQL is 1.1–3 times faster than RxJava, 1.5–15 times faster than Reactor, and 5–50 times faster than Siddhi.

3.7.6 TAQ Benchmark

In the TAQ benchmark, we used data from the NYSE TAQ database [80], which collects real-time *trades* and *quotes* reported on the U.S. Consolidated Tape (where billions of entries are recorded per day). We implemented the following queries: ***T1*** filters out events that are outside normal NYSE hours, ***T2*** computes the running average price for each stock, ***T3*** computes the average price for each stock over a tumbling window, ***T4*** (***T5***) computes the sequence of trading intervals (durations between consecutive transactions) for a specific (each) stock, ***T6*** counts the number of odd lots (trades with less than 100 shares) for each stock, ***T7*** (***T8***) computes the

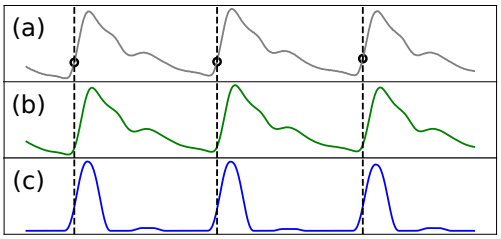


Figure 3.14 : Examples of (a) raw ABP signal with onset labels, (b) low-pass filtered signal, and (c) SSF signal.

best bid and offer for a specific (each) stock over a tumbling window, and ***T9*** (***T10***) calculates the true value estimate for a specific (each) stock over a tumbling window. Figure 3.13 shows the results: StreamQL is 1.2–2 times faster than RxJava, 2–10 times faster than Reactor, and 8–100 times faster than Siddhi.

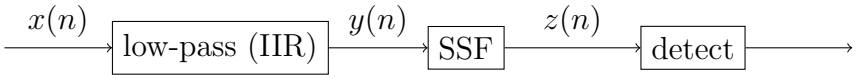
3.8 Case Study: Arterial Blood Pressure Monitoring

We will use StreamQL to specify a streaming algorithm for Arterial Blood Pressure (ABP) pulse detection [222, 81]. This is a complex streaming computation and is difficult to express with existing languages for stream processing. The use of a streaming query language for medical monitoring applications has been considered in [223, 224]. StreamQL, with its design for expressing high-level specifications, makes it an ideal choice for this task.

The ABP signal is collected from the MIT-BIH Polysomnographic database [4]. The signal measurements are of type $\texttt{VT} = \{\texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}\}$, where `val` is the value of the signal and `ts` is the timestamp. The signal is uniformly sampled at a frequency of 250 Hz. Figure 3.14 shows a snippet of an ABP signal containing 3 ABP pulses (around 3 seconds). The ABP waveform contains rich information about the cardiovascular system (e.g., heart rate, systolic, mean, and diastolic arterial pressures). Reliable

ABP pulse detection is crucial for extracting this information.

First, the algorithm preprocesses the signal stream using a low-pass IIR filter and a slope sum function (SSF), and then it performs the detection of the pulse onset.



The low-pass filter suppresses high frequency noise, and is defined by $y(n) = 2y(n-1) - y(n-2) + x(n) - 2x(n-5) + x(n-10)$. The SSF is defined by $z(n) = \sum_{0 \leq i \leq 31} max(0, d(n-i))$, where $d(n) = y(n) - y(n-1)$. It enhances the up-slope of the ABP pulse and restrains the remainder of the pressure waveform. The query $\texttt{getVTP} : \texttt{Q}(\texttt{VT}, \texttt{VTP})$ annotates each item $\{\texttt{val}, \texttt{ts}\}$ of the input stream with an additional component `pval`, which is the result of the preprocessing. The type $\texttt{VTP} = \{\texttt{val} : \texttt{V}, \texttt{ts} : \texttt{T}, \texttt{pval} : \texttt{V}\}$ extends `VT` with this additional component. These preprocessed values have a phase shift of 20 ms (5 samples), which is introduced by low-pass filtering.

The detection of ABP onset is described by the following rules: **R1.** In intervals where the SSF value exceeds a threshold `Thred` (i.e. a tentative pulse), the algorithm selects the first and the maximum SSF values. **R2.** The pulse detection is accepted only if the difference between the first and the maximum SSF values exceeds 100. **R3.** When the pulse is accepted, the algorithm chooses the first sample that crosses the threshold as the onset point. The detected onset is adjusted by 20 ms (5 samples) to compensate for the phase shift of low-pass filtering. **R4.** After an onset is detected, to avoid double detection of the same pulse, the detection falls silent for 300 ms. Figure 3.15 shows the StreamQL implementation of the detection algorithm.

```
# preprocess the signal
lowPass = IIR({−1, 2}, {1, 0, 0, 0, 0, −2, 0, 0, 0, 0, 1})
diff = sWindow(2, 1, (x, y) -> y − x)
sum = sWindow(32, 1, reduce((x, y) -> (y > 0) ? (x + y) : x))
ssf = diff ≫ sum
preProc = map(x -> x.val) ≫ lowPass ≫ ssf
getVTP = annotate(preProc, (x, y) -> ⟨x.val, x.ts, y⟩)
# select signal interval containing a peak (R1)
pulse = takeWhen(x -> x.pval > Thred, x -> x.pval < Thred)
# select the first element in interval as the onset sample
# find the measurement with the maximum preprocessed value,
# and store them as a pair ⟨first, max⟩
select = reduce(x -> ⟨x, x⟩, (⟨f, m⟩, x) -> ⟨f, (x.pval > m.pval) ? x : m⟩)
# examine the detected pulse (R2) and project the onset
getOnset = filterMap(⟨f, m⟩ -> m.pval − f.pval > 100, ⟨f, m⟩ -> f)
detect1 = getVTP ≫ pulse ≫ select ≫ check ≫ getOnset
rft = skip(75)  # after detecting the ABP onset, apply R4
detectAll = seq(detect1, iter(rft ≫ detect1))
subShift = map(x -> x.ts − 5)  # compensate for phase shift
ABPDetection = detectAll ≫ subShift
```

Figure 3.15 : StreamQL program for ABP pulse detection.

3.9 Chapter Summary

We have introduced StreamQL, a language that specifies complex streaming computations as combinations of stream transformations. StreamQL integrates relational, dataflow, and temporal language constructs to provide an expressive and modular high-level approach for programming streaming analyses. We have implemented StreamQL as a Java library, and we have compared its performance against three popular streaming engines (RxJava, Reactor, and Siddhi) using four benchmarks. In benchmarking with real-world applications, the throughput of the StreamQL library is consistently higher: 1.1–10 times higher than RxJava, 1.2–20 times higher than Reactor, and 5–100 times higher than Siddhi. We have used StreamQL to easily prototype a streaming algorithm for ABP (Arterial Blood Pressure) pulse detection, a complex computation that is difficult to express in other streaming languages.

Chapter 4

Recognition of Regular Patterns

4.1 Motivation

Regular pattern matching, where the patterns are expressed with finite-state automata or regular expressions, has numerous applications in text search and analysis [151], network security [43], bioinformatics [65, 66], and runtime verification [44, 67]. Various techniques have been developed for matching regular patterns, many of which are based on the execution of deterministic finite automata (DFAs) or nondeterministic finite automata (NFAs). DFA-based techniques are generally faster, as the processing of an input element requires a single memory lookup, while NFA-based techniques are slower, as they involve extending several execution paths when processing one element. The advantage of NFAs over DFAs is that they are typically more memory-efficient, and there are cases where an equivalent DFA would unavoidably be exponentially larger [69].

Many applications require the processing of large and complex NFAs on real-time streams of data collected from sensors, networks, and various system traces. Energy efficiency and memory efficiency (in terms of the memory capacity or chip footprint needed for a given NFA) are highly desirable for both high-performance computing and battery-powered embedded applications. NFA processing requires frequent, yet irregular and unpredictable, memory accesses on general-purpose processors, leading to limited throughput and high power on CPU and GPU architectures [70, 71, 72].

Field Programmable Gate Arrays (FPGAs) offer high speed through hardware-level parallelism, but are often bottlenecked by routing congestion [73, 74] and their high power, area and cost prevent their use in mobile and embedded devices. Even with digital application-specific integrated circuit (ASIC) accelerators, the memory access bandwidth restricts the parallelism [75, 76]. The latest hardware technology that addresses these challenges is in-memory architecture. This architecture processes the NFA transitions directly inside memories with massive parallelism and provides high throughput. For instance, the Automata Processor (AP) from Micron [54, 77] outperforms x86 CPUs by 256 times and general purpose GPUs by 32 times in the ANMLZoo benchmark suite [70, 225].

Classical regular expressions (regexes) involve operators for concatenation $\cdot$, nondeterministic choice $+$, and iteration (Kleene's star) * (see more details in Section 2.2.1). They can be translated into NFAs whose size is linear in the size of the regex [78, 79]. However, the regexes used in practice have several additional features that make them more succinct. One such feature is *counting*, written as $r\{m, n\}$, which is also called *bounded repetition*. The pattern $r\{m, n\}$ expresses that the subpattern r is repeated anywhere from m to n times. This bounded repetition is ubiquitous in practical use cases of regexes. For instance, in our analysis of various datasets pertaining to network intrusion detection, such as Snort [133] and Suricata [134], as well as motif searches in biological sequences like Protomata [135, 65], we found the occurrence of bounded repetitions in the majority of the patterns. The naïve approach for dealing with counting operators is to rewrite them by *unfolding*. For example, $r\{n, n\}$ is unfolded into $r \cdot r \cdots r$ (n-fold concatenation) and results in an NFA of size linear in n (and therefore can produce a DFA of size exponential in n). Given that n can scale significantly, handling counting emerges as one of the main technical hurdles for the

effective implementation of hardware-based approaches in executing practical regular patterns.

Existing in-memory NFA architectures use this naïve unfolding method to handle bounded repetition. This leads to the use of a large number of STEs* to support counting. In AP [54] and CA (Cache Automaton) [225], each STE uses 256 memory bits for 8-bit symbols. In the latest Impala [226] and CAMA† [53] designs, each STE requires 16 to 32 memory bits. Even with this improvement, a modest counting operator with an upper limit 1024 requires at least 16384 memory bits, while the information required for implementing the operator may be only 10 bits in some cases. Therefore, the unfolding solution results in large memory and energy cost.

4.2 Contributions

We explored software and hardware co-design for integrating counter and bit vector modules into a state-of-the-art in-memory NFA architecture. Our design is inspired by an extension of NFAs with counter registers called nondeterministic counter automata (NCAs). In an NCA, a computation path involves not only transitions between control states, but also the use of a finite number of registers that hold nonnegative integers. Such automata are a natural execution model for regexes with counting, as the counters can track the number of repetitions of subpatterns. When the counters are bounded, NCAs are expressively equivalent to NFAs, but they can be exponentially more succinct [69, 227]. Similar to how an NFA is executed by maintaining the set of active states, an NCA is executed by maintaining a set of pairs, which we call *tokens*,

*STE stands for State Transition Element [54]. It is a hardware element that roughly corresponds to the state of a homogeneous NFA. It contains a state bit (to indicate whether the state is active or not) and a memory array that represents a character class.

†CAMA abbreviates Content Addressable Memory (CAM) enabled Automata accelerator.

where the first component is the control state and the second component specifies the values of the counters. A key idea of our approach is that we can statically analyze an NCA to determine which states can carry a large number of tokens during execution. We call a control state *counter-unambiguous* if it can only carry at most one token and *counter-ambiguous* if it can carry more than one. In the case of counter-unambiguity for a state q with counter x, we know that we only need to record one counter value, which means that we need only one memory location whose size (in bits) is logarithmic in the range M of possible counter values. In the case of counter-ambiguity for q with counter x, we may have to record a large number of counter values (as large as M), and our insight is to use a bit vector v of size M, where $v[i] = 1$ (resp., $v[i] = 0$) indicates the presence (resp., absence) of a token at q with counter value i. Therefore, identifying a state as counter-unambiguous enables a massive memory reduction for this state from $O(M)$ to $O(\log M)$.

We designed a *static analysis algorithm* for checking the counter-ambiguity of NCAs and regexes by performing a systematic exploration of the space of reachable tokens to identify the existence of some input string for which two different tokens are placed on the same control state. This may lead to a large search space (exponential in the size of the regex), and the worst case is not easy to avoid since the problem is NP-hard. To handle difficult instances that involve large repetition bounds, we also designed an *over-approximate* algorithm that gives an inconclusive output for some instances, while still being able to identify cases of counter-unambiguity for most cases in five real-world benchmarks. By combining the exact and over-approximate algorithms, we can statically analyze within milliseconds the vast majority of regexes.

Using the insights about NCA execution mentioned earlier, we proposed a hardware design that is based on existing in-memory NFA architectures (AP, CA, Impala,

CAMA) augmented with (1) *counter modules* for counter-unambiguous states, and (2) *bit vector modules* for counter-ambiguous states. We also provided a compiler that statically analyzes an input regex to determine counter-(un)ambiguity and then creates a representation of an automaton with counters and bit vectors using the MNRL format [228] that can be used to program the hardware. Several existing architectures like AP provide a counter module in their design, but they typically do not provide a compiler that translates regexes to hardware-recognizable programs. Also, counter registers alone cannot deal with the challenging instances of counting. Compared with prior works that do not provide a bit vector module, we proposed a novel design that can systematically handle counting and ensure correct compilation in both the easy (requiring counters) and difficult (requiring bit vectors) cases.

We modified the open-source simulator VASim [70] to simulate the hardware performance of our counter- and bit-vector-augmented CAMA design. In microbenchmarks, we evaluated the energy and area consumption of counters and bit vectors against their unfolded counterparts. The results show that our counter- and bit-vector-based design can reduce energy usage by orders of magnitude and the area by large margins. Furthermore, we evaluated the performance of the augmented CAMA design using the Snort [133], Suricata [134], Protomata [65], and SpamAssassin [229] benchmarks. For applications involving regexes with large counting upper bounds[‡], the results show as large as 76% energy reduction and 58% area reduction. For regexes with small counting upper bounds, the results show little to no overhead.

The main contributions are summarized below:

- We use the notion of *counter-unambiguity* to identify instances of bounded

[‡]For the patterns $r\{n\}$ (which is the abbreviation for $r\{n,n\}$), $r\{m,n\}$, and $r\{n,\}$ (which is equivalent to $r\{n\} \cdot r^*$), the value n is the upper bound of counting.

repetition that can be handled with a small amount of memory. We describe both an exact and an over-approximate *static analysis* for counter-(un)ambiguity which, when combined, allow us to efficiently analyze the regexes that arise in several application domains.

- We provide a *compiler* that enables the high-level programming of the hardware using POSIX-style regexes. The compiler first performs the static analysis for counter-(un)ambiguity and then leverages the analysis results for producing a low-level description of the automaton.

- We propose a *hardware design* that augments the prior NFA-based CAMA architecture [53] with counter and bit vector modules, which are inspired from the execution of NCAs and the classification of states as counter-(un)ambiguous. This architecture achieves substantial energy and area reductions compared to prior designs.

4.3 Nondeterministic Counter Automata

Nondeterministic counter automata (NCAs) is an extension of NFAs with counter registers. In an NCA, a computation path involves not only transitions between control states, but also the use of a finite number of registers that hold nonnegative integers. Such automata are a natural execution model for regexes with counting, as the counters can track the number of repetitions of subpatterns. When the counters are bounded, NCAs are expressively equivalent to NFAs, but they can be exponentially more succinct [69, 227].

We fix an infinite set *CReg* of counter registers or, simply, *counters*. We typically write $x, y, z, \ldots$ to denote counter registers. For a subset $V \subseteq \mathit{CReg}$ of counters, we

say that a function $\beta : V \to \mathbb{N}$, which assigns a value to each counter in V, is a *V-valuation*.

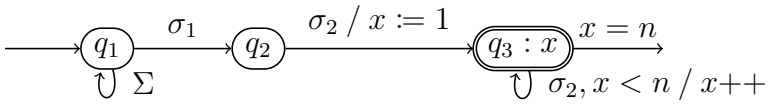
Definition 4.3.1. Let Σ be a finite alphabet. A nondeterministic counter automaton (NCA) with input alphabet Σ is a tuple $\mathcal{A} = (Q, R, \Delta, I, F)$, where

- Q is a finite set of *states*,
- $R : Q \to \mathcal{P}(CReg)$ is a function that maps each state to a finite set of counters,
- Δ is the *transition relation*, which contains finitely many transitions of the form $(p, \sigma, \varphi, q, \vartheta)$, where p is the source state, $\sigma \subseteq \Sigma$ is a predicate over the alphabet, $\varphi \subseteq (R(p) \to \mathbb{N})$ is a predicate over $R(p)$-valuations, q is the destination state, and $\vartheta : (R(p) \to \mathbb{N}) \to (R(q) \to \mathbb{N})$,
- I is the *initialization function*, a partial function defined on the subset $\mathrm{dom}(I) \subseteq Q$ of *initial states* that specifies an *initial valuation* $I(q) : R(q) \to \mathbb{N}$ for each initial state q, and
- F is the *finalization function*, a partial function defined on the subset $\mathrm{dom}(F) \subseteq Q$ of *final states* that specifies a predicate $F(q) \subseteq R(q) \to \mathbb{N}$ for each final state q.

We remark that the states in an NCA defined above do not necessarily have the same counters. In fact, some states may not have any counter at all, and we say that a state $q \in Q$ is *pure* if $R(q) = \emptyset$, that is, it has no counter associated with it. In a transition $(p, \sigma, \varphi, q, \vartheta)$, we will call the predicate φ a *guard* because it may restrict a transition based on the values of the counters, and we will call the function ϑ an *action* because it describes how to assign counter values to the destination state given the counter values in the source state.

We convert regexes (with counting) to NCAs that recognize the same language using a variant of the Glushkov construction [79, 230]. In contrast to Thompson's construction [78], Glushkov's construction results in ε-free automata that are also *homogeneous*, i.e., all incoming transitions of a state are labeled with the same predicate over the alphabet.

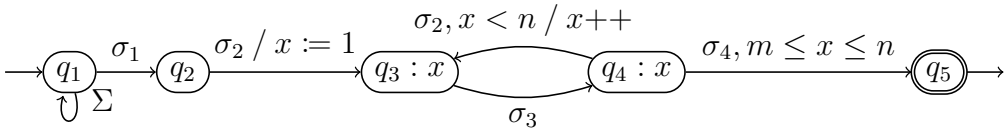
Example 4.3.1. Consider the regex $r_1 = \Sigma^*\sigma_1\sigma_2\{n\}$ with $n \geq 1$, where σ_1, σ_2 are predicates over the alphabet. The following automaton recognizes the language of r_1:



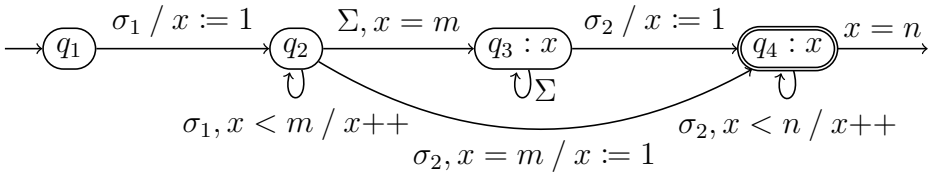
The automaton above has three states: q_1, q_2, and q_3. We write $q_3 : x$ to indicate that $R(q_3) = \{x\}$. Notice that q_1 has no annotation with counters, which means that $R(q_1) = \emptyset$ (i.e., q_1 is pure). We annotate each edge $p \to q$ with an expression of the form $\sigma, \varphi \,/\, \vartheta$, where σ is a predicate over Σ, φ is a guard over the counters of p, and ϑ is an assignment for the counters of q using the counters of p. If the guard φ is omitted, then it is always true. The action ϑ is omitted only when $R(q) \subseteq R(p)$, and the omission indicates that the counters $R(q)$ retain the values from the previous state. We can also indicate this explicitly by writing "$x \coloneqq x$". We write "$x = n$" for the guard that checks whether the value of counter x is equal to n, and we write "$x \coloneqq n$" to denote the assignment (action) of the value n to the counter x. We use double circle notation to indicate that a state is final (see state q_3 above). An arrow emanating from a final state q is annotated with the predicate $F(q)$ over counter valuations (recall that F is the finalization function).

The regex $r_2 = \Sigma^*\sigma_1(\sigma_2\sigma_3)\{m, n\}\sigma_4$ with $1 \leq m \leq n$ is recognized by the following

automaton:



The regex $r_3 = \sigma_1\{m\}\Sigma^*\sigma_2\{n\}$ with $m, n \geq 1$ is recognized by the automaton below:



Nondeterministic Semantics of NCA

Let $\mathcal{A}$ be an NCA. A *token* for $\mathcal{A}$ is a pair (q, β), where q is a state and $\beta : R(q) \to \mathbb{N}$ is a counter valuation for q. The set of all tokens for $\mathcal{A}$ is denoted by $\mathbf{Tk}(\mathcal{A})$. For a letter $a \in \Sigma$, we define the *token transition relation* $\to^a$ on $\mathbf{Tk}(\mathcal{A})$ as follows: $(p, \beta) \to^a (q, \gamma)$ if there is a transition $(p, \sigma, \varphi, q, \vartheta) \in \Delta$ with $a \in \sigma$ such that $\beta \in \varphi$ and $\gamma = \vartheta(\beta)$. A token (q, β) is *initial* if the state q is initial. A token (q, β) is *final* if the state q is final and $\beta \in F(q)$. A *run* of $\mathcal{A}$ on a string $a_1 a_2 \dots a_n \in \Sigma^*$ is a sequence

$$(q_0, \beta_0) \xrightarrow{a_1} (q_1, \beta_1) \xrightarrow{a_2} (q_2, \beta_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q_n, \beta_n),$$

where each (q_i, β_i) is a token, q_0 is an initial state and $\beta_0 = I(q_0)$, and $(q_{i-1}, \beta_{i-1}) \to^a (q_i, \beta_i)$ for every $i = 1, \dots, n$. A run is *accepting* if it ends with a final token. The NCA $\mathcal{A}$ *accepts* a string if there is an accepting run on it. We write $[\![\mathcal{A}]\!] \subseteq \Sigma^*$ for the set of strings that $\mathcal{A}$ accepts.

Notice that, for a NCA $\mathcal{A}$, the set of tokens $\mathbf{Tk}(\mathcal{A})$ together with the transition relations $\to^a$ forms a labeled transition system. The family of transition relations $(\to^a)_{a\in\Sigma}$ can be represented as a ternary relation $\to \subseteq \mathbf{Tk}(\mathcal{A}) \times \Sigma \times \mathbf{Tk}(\mathcal{A})$.

For a pure state q (i.e., a state with no counter, see Definition 4.3.1), there is only one valuation, denoted $0_{\mathbb{N}} : \emptyset \to \mathbb{N}$, which carries no information. So, we will often abuse notation and simply write q for the token $(q, 0_{\mathbb{N}})$. Similarly, for a state q with one counter, i.e., $R(q) = \{x\}$ for some $x \in \mathit{CReg}$, a valuation β (of type $\{x\} \to \mathbb{N}$) for q specifies only one value $c = \beta(x)$ for the unique variable x for q. For this reason, we will sometimes write (q, c) for a token for the state q.

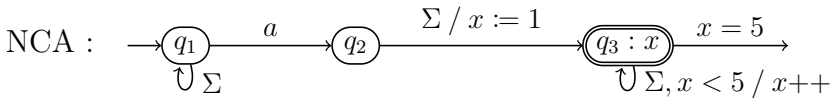
Semantics Using NCA Configurations

Let $\mathcal{A}$ be an NCA. A *configuration* for $\mathcal{A}$ is a set of tokens for $\mathcal{A}$. We write $\mathbf{C}(\mathcal{A})$ for the set of all configurations for $\mathcal{A}$. Define the configuration transition function $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma \to \mathbf{C}(\mathcal{A})$ as follows:

$$\delta(S, a) = \{(q, \gamma) \mid (p, \beta) \to^a (q, \gamma) \text{ for some } (p, \beta) \in S\}.$$

We extend the transition function to $\delta : \mathbf{C}(\mathcal{A}) \times \Sigma^* \to \mathbf{C}(\mathcal{A})$ by $\delta(S, \varepsilon) = S$ and $\delta(S, xa) = \delta(\delta(S, x), a)$ for every $x \in \Sigma^*$ and $a \in \Sigma$. Let S_0 be the set of all initial tokens, which we call the *initial configuration*, and define $[\mathcal{A}] : \Sigma^* \to \mathbf{C}(\mathcal{A})$ by $[\mathcal{A}](x) = \delta(S_0, x)$. This semantics coincides with $[\![\mathcal{A}]\!]$ in the following sense: for every $x \in \Sigma^*$, $x \in [\![\mathcal{A}]\!]$ iff $[\mathcal{A}](x)$ contains some final token.

Example 4.3.2. Fig. 4.1 shows the execution of the NCA for the regex $\Sigma^* a \Sigma\{5\}$, where a is a predicate that returns true if the input item is character a. In this figure, q_1 (resp., q_2) is the abbreviation for the token $(q_1, 0_{\mathbb{N}})$ (resp., $(q_2, 0_{\mathbb{N}})$), i.e., q_1 (resp.,



input	NCA configuration	output
	q_1	0
b	q_1	0
a	q_1, q_2	0
b	$q_1, (q_3, 1)$	0
a	$q_1, q_2, (q_3, 2)$	0
a	$q_1, q_2, (q_3, 1), (q_3, 3)$	0
b	$q_1, (q_3, 1), (q_3, 2), (q_3, 4)$	0
a	$q_1, q_2, (q_3, 2), (q_3, 3), (q_3, 5)$	1
a	$q_1, q_2, (q_3, 1), (q_3, 3), (q_3, 4)$	0
a	$q_1, q_2, (q_3, 1), (q_3, 2), (q_3, 4), (q_3, 5)$	1
a	$q_1, q_2, (q_3, 1), (q_3, 2), (q_3, 3), (q_3, 5)$	1
b	$q_1, (q_3, 1), (q_3, 2), (q_3, 3), (q_3, 4)$	0

Figure 4.1 : Execution of the NCA for the regular expression $\Sigma^* a \Sigma\{5\}$.

q_2) is a pure state. The notation (q_i, n) is the abbreviation for the token $(q_i, x \mapsto n)$ (the counter assignment maps x to n).

4.4 Static Analysis over Regular Expressions

In this section, we will see how to perform a static analysis over regexes to check counter-(un)ambiguity. It is well-known that the presence of counting in regexes can cause a blow-up in the amount of memory that is needed for the streaming membership problem (checking if a string matches the regex in a single left-to-right pass) [69, 231, 227]. There are, however, many cases that do not exhibit this worst-case behavior. In this section, we will describe a static analysis for identifying occurrences of bounded repetition $\{m, n\}$ which can be implemented using memory that is logarithmic in n. This enables a significant reduction in the memory that

needs to be reserved for the membership problem. In order to identify the easier cases of bounded repetition, we use the concept of counter-unambiguity, which informally says that the nondeterminism of the automaton is constrained. We then develop two algorithms for deciding counter-unambiguity (one exact and one approximate), and we provide experimental results showing that they are effective in practice.

Let $\mathcal{A} = (Q, R, \Delta, I, F)$ be an NCA. For a state $q \in Q$ and a subset $T \subseteq \mathbf{Tk}(\mathcal{A})$ of tokens for the automaton, define $T|_q = T \cap (\{q\} \times (R(q) \to \mathbb{N}))$. That is, $T|_q$ contains exactly those tokens of T whose first component is the state q. The operational intuition is that $[\mathcal{A}](x)|_q$ is the set of tokens that we get at state q when we execute the automaton $\mathcal{A}$ on input x. When it is possible to have more than two tokens on the same state q after consuming an input string, we say that the state exhibits *counter-ambiguity*. We will now define this concept and other related notions more formally.

Definition 4.4.1 (**Counter-Ambiguity).** Let $\mathcal{A}$ be an NCA with bounded counters and q be a state. The *counter-ambiguity degree* of q is defined as

$$\mathsf{degree}(q) = \sup_{x \in \Sigma^*} \big(\text{size of } [\mathcal{A}](x)|_q\big).$$

We say that q is *counter-unambiguous* when $\mathsf{degree}(q) \leq 1$, and that q is *counter-ambiguous* when $\mathsf{degree}(q) \geq 2$.

4.4.1 Deciding Counter-Ambiguity

According to Definition 4.4.1, the degree of counter-ambiguity of a state q is the maximum number of different tokens that can end up at q during a computation. A state q is counter-ambiguous iff there is a string $a_1 a_2 \ldots a_n \in \Sigma^*$ and two different

runs on $a_1 a_2 \ldots a_n$

$$(q_0, \beta_0) \xrightarrow{a_1} (q_1, \beta_1) \xrightarrow{a_2} (q_2, \beta_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q_n, \beta_n)$$
$$(q'_0, \beta'_0) \xrightarrow{a_1} (q'_1, \beta'_1) \xrightarrow{a_2} (q'_2, \beta'_2) \xrightarrow{a_3} \cdots \xrightarrow{a_n} (q'_n, \beta'_n),$$

such that $q = q_n = q'_n$ and $\beta_n \neq \beta'_n$.

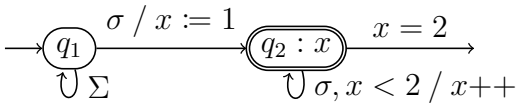
Let G be the labeled transition system of tokens $\mathbf{Tk}(\mathcal{A})$ and token transitions of the form $t_1 \rightarrow^a t_2$, where t_1, t_2 are tokens and $a \in \Sigma$. Define $G^2 = G \times G$ to be the *product* transition system with states $\mathbf{Tk}(\mathcal{A}) \times \mathbf{Tk}(\mathcal{A})$, which contains a transition $\langle t_1, t_2 \rangle \rightarrow^a \langle t'_1, t'_2 \rangle$ iff $t_1 \rightarrow^a t'_1$ and $t_2 \rightarrow^a t'_2$. A pair $\langle t_1, t_2 \rangle$ is initial if both t_1 and t_2 are initial tokens. According to the characterization of the previous paragraph, a state q of $\mathcal{A}$ is counter-ambiguous iff there exists a path in G^2 that ends with some pair $\langle (q, \beta), (q, \beta') \rangle$, where $\beta \neq \beta'$. This idea can be extended to characterize the situation where a state q has degree at least $d \geq 2$: there exists a path in the d-fold Cartesian product G^d that ends with some tuple $\langle (q, \beta_1), \ldots, (q, \beta_d) \rangle$, where $\beta_1, \ldots, \beta_d$ are all distinct.

Algorithm for Determining Counter-Ambiguity

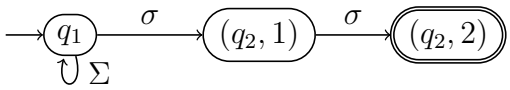
When the product transition system G^d is finite, we can decide whether the counter-ambiguity degree of a state is $\geq d$ with a straightforward reachability algorithm. For deciding counter-ambiguity, we check whether the degree is ≥ 2, and therefore it suffices to consider only G^2. Notice that for the bounded counter automata that we consider, G^d is always finite. We just need to exercise care to avoid a blowup in the number of transitions. In our automata, the transitions are annotated with predicates over the alphabet, not symbols of the alphabet. This is a succinct way to

represent transitions, and we want to maintain such a representation in the graphs G^d (assuming that we also use such a representation for G). This can be done by considering the intersections of predicates and checking whether they are empty. More specifically, for every pair of transitions $t_1 \to^{\sigma_1} t_1'$ and $t_2 \to^{\sigma_2} t_2'$, we add the transition $\langle t_1, t_2 \rangle \to^{\sigma_1 \cap \sigma_2} \langle t_1', t_2' \rangle$ in G^2 when $\sigma_1 \cap \sigma_2$ is nonempty.

Example 4.4.1. We will discuss here how to check counter-(un)ambiguity for the regex $\Sigma^* \sigma\{2\}$. First, we construct the NCA for this regex, which is seen below:



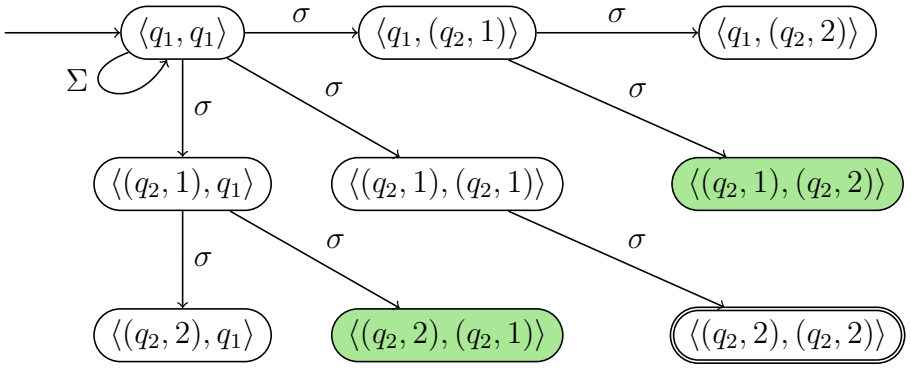
Based on this NCA, we construct the transition system of tokens seen below:



The token transition system is essentially an NFA, where the final state (token) is indicated with a double circle.

To check the counter-ambiguity of a state q, we build the product transition system and check whether there exists a path that ends in a pair of tokens $\langle (q, \beta), (q, \beta') \rangle$

with $\beta \neq \beta'$. The figure below shows the product transition system:



In the figure above, the presence of the pair $\langle (q_2, 1), (q_2, 2) \rangle$ or $\langle (q_2, 2), (q_2, 1) \rangle$ (colored in green) witnesses the counter-ambiguity. Because of symmetry, some states and transitions can be safely removed from the product automaton. Notice, for example, that we do not need to explore both $\langle (q_2, 1), q_1 \rangle$ and $\langle q_1, (q_2, 1) \rangle$. Therefore, in future examples, we will omit part of the product automaton.

The exact analysis halts as soon as it finds a token pair that witnesses counter-ambiguity. So, not all pairs are generated during the static analysis, unless the regex is counter-unambiguous.

Consider a regex r that contains an occurrence of counting of the form $(abcd)\{m, n\}$. When the repetition bounds are sufficiently large, in the automaton $\mathcal{A}$ for r, the four states that correspond to *abcd* are either all counter-unambiguous or they are all counter-ambiguous. For this reason, the notion of counter-(un)ambiguity can be defined with respect to instances of bounded repetition in regexes. We will also call a regex counter-ambiguous if it contains at least one occurrence of bounded repetition that is counter-ambiguous (equivalently, the NCA for the expression has at least one counter-ambiguous state).

Lemma 4.4.1 (Checking Counter-Ambiguity Is Hard). Let CAMBIGUITY be the following problem: Given a regex r as input, is r counter-ambiguous? CAMBIGUITY is NP-hard.

Proof. Consider the alphabet $\Sigma = \{a, b, \#\}$. We will give a polynomial-time reduction from the subset sum problem to CAMBIGUITY. Let $S = \{n_1, n_2, \dots n_m\}$ be a set of natural numbers and T be a natural number. Recall that the subset sum problem asks whether there is a subset $S' \subseteq S$ of numbers whose sum is equal to T. Consider the regex

$$(((a\{n_1\} + \varepsilon) \cdots (a\{n_m\} + \varepsilon)\#b) + (a\{T\}\#bb))b\{2\}.$$

We focus on the rightmost occurrence of bounded repetition (i.e., $b\{2\}$). We claim that this occurrence is counter-ambiguous if and only if there is a subset $S' \subseteq S$ whose sum is T. Consider the corresponding Glushkov automaton and the state q which leads to the final state at the end that recognizes the $b\{2\}$. A word witnessing a path to q would have to be of the form $a^x\#b^y$ for some natural numbers x, y. If $x \neq T$, then the word has no path through the branch $(a\{T\}\#bb)$. So, the only value it can induce on the counter at the end is $(y - 2)$. If $x = T$, and there exists a subset S' of S such that $\sum S' = T$, then $a\{T\}\#bbb$ could either take the path $(a\{T\}\#bb)$ and set the counter to 1, or it could take the other path and set the counter to 2. If $x = T$ and there is no such subset S', then the only path the word can take is through the branch $(a\{T\}\#bb)$ which would set the counter to $(y - 2)$. □

4.4.2 Over-Approximate Analysis

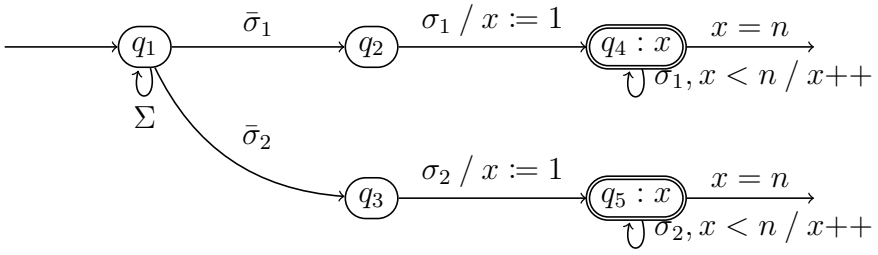
In Section 4.4.1, we presented an (exact) algorithm for deciding the counter-(un)ambiguity of regexes and NCAs. The algorithm operates on the transition system of tokens

of an NCA, whose size can be exponential in the size of the regex, because of the counter valuations. For example, the regex $\Sigma^* \cdot a \cdot \Sigma\{n\}$ has size $\Theta(\log n)$ (because the repetition bound n is represented succinctly in binary or decimal notation) and the corresponding token transition system has size $\Theta(n)$. From this it follows that the exact algorithm may need exponential time in the worst case. Unfortunately, this worst-case behavior is not easy to avoid given the NP-hardness of the problem (Lemma 4.4.1). For this reason, we propose here a heuristic algorithm that performs an "over-approximate" analysis, which can give two outputs: it either declares that a state is counter-unambiguous, or it says that the analysis is inconclusive. In other words, there are cases where the algorithm may suspect that a state is counter-ambiguous, but it cannot conclusively declare it so.

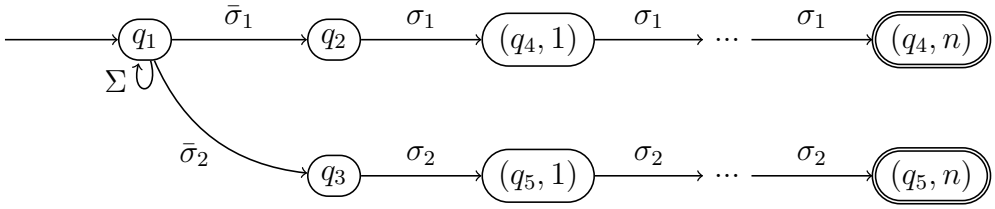
The idea is to over-approximate all occurrences of $\{m, n\}$ (constrained repetition) with * (unconstrained repetition), except for the one that we are analyzing. If we think of this transformation in terms of NCAs, we see that it adds more paths to the token transition graph, because more transitions are now enabled. A consequence of this is that if the over-approximate automaton is counter-unambiguous, then surely the original automaton (which has less paths) is also counter-unambiguous. On the other hand, if the over-approximate automaton is counter-ambiguous, then we cannot infer that the original automaton is counter-ambiguous.

Example 4.4.2. We show the static analysis for a counter-unambiguous regex $r = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2\{n\})$, where n is a constant. For this regex, the over-approximate analysis is more efficient than the exact analysis. To illustrate this, we first construct

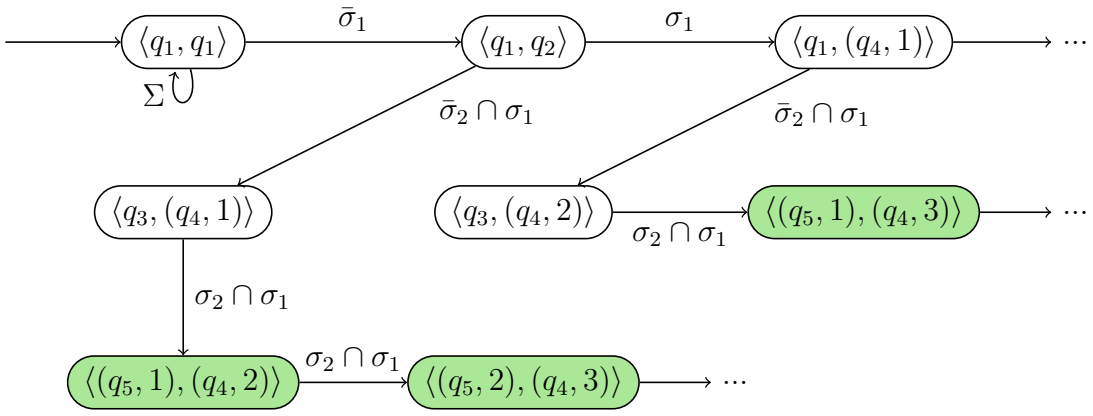
the following NCA:



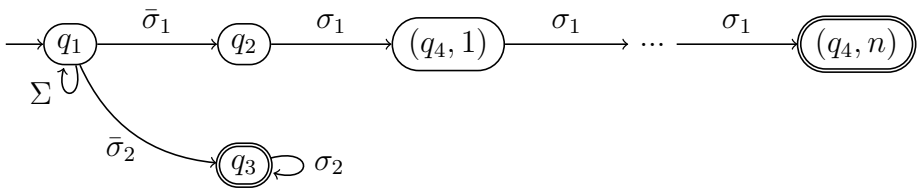
The exact analysis constructs the token transition system:



To determine whether the regex is counter-unambiguous, the exact analysis explores all possible token pairs in the product transition system. In this example, the number of explored pairs is $\Theta(n^2)$. Below is a part of the product transition system, in which all token pairs $\langle (q_5, i), (q_4, j) \rangle$ with $1 \leq i < j \leq n$ (colored in green) will be explored.



We observe that regexes of the form $r = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2\{n\})$, where n is a large number, can be found in the Snort and Suricata benchmarks. For these regexes, the exact analysis may require a long computation. Fortunately, the over-approximate analysis is substantially faster. We approximate the regex as $r' = \Sigma^*(\bar{\sigma}_1\sigma_1\{n\} + \bar{\sigma}_2\sigma_2^*)$ and $r'' = \Sigma^*(\bar{\sigma}_1\sigma_1^* + \bar{\sigma}_2\sigma_2\{n\})$ and check the counter-ambiguity of r' and r'' using the exact analysis. The regex r is determined to be counter-unambiguous if both r' and r'' are counter-unambiguous. Below, we construct the token transition system G for r'. Only $\Theta(n)$ token pairs are explored in the product transition system G^2.



The over-approximate analysis checks the counter-ambiguity of r', r''. So, it reduces the complexity from $\Theta(n^2)$ to $\Theta(n)$.

NCA Execution with Bit Vectors

If the static analysis determines that an NCA state q is counter-ambiguous, then this implies that the execution of the automaton may require several memory locations to store tokens of the form (q, β). Assuming that q has only one counter register x (i.e., $R(q) = \{x\}$) and that q is n-bounded, we know that there are at most n different possible tokens. In order to compactly represent a set of tokens, the idea is to use a bit vector that indicates the presence or the absence of a specific token on q. So, a bit vector v encodes a set of tokens on q as follows: $v[i] = 1$ iff the token (q, i) is active. We can also think of a bit vector as a representation for part of the automaton

configuration (recall the configuration semantics from Section4.3).

It remains to see how the execution of the automaton can be described using these bit vectors to represent the configuration. Example 4.3.1 shows the NCA for the regex $\Sigma^*\sigma_1(\sigma_2\sigma_3)\{m,n\}\sigma_4$. This NCA is general enough to illustrate the main ways in which we manipulate bit vectors:

(1) Consider a transition $p \to q$, annotated with "$\sigma \,/\, x \coloneqq c$", where p is pure and $R(q) = \{x\}$. A token on p is transformed into a bit vector v for q that is everywhere 0 except that $v[c] = 1$.

(2) Let $p \to q$ be a transition, annotated with σ, where $R(p) = R(q) = \{x\}$. Since the transition does not change the counter valuations, a bit vector v on p is passed along unchanged to q.

(3) We will deal now with a transition $p \to q$, annotated with "$\sigma, x < n \,/\, x{+}{+}$", where $R(p) = R(q) = \{x\}$. Assume further that both p and q are n-bounded, which means that each state carries a bit vector of size n. This transition corresponds to performing a *shift operation* to the bit vector v of p, resulting in a new bit vector v' for q. We have: $v'[1] = 0$ and $v'[i+1] = v[i]$ for ever $i = 2, \ldots, n-1$.

(4) Finally, let us consider a transition $p \to q$, annotated with "$\sigma, m \leq x \leq n$", where $R(p) = \{x\}$ and q is pure. If v is the current bit vector for p, then taking this transition produces a token for q if and only if one of $v[m], v[m{+}1], \ldots, v[n{-}1], v[n]$ is equal to 1. In other words, we have to compute the disjunction $v[m] \vee \cdots \vee v[n]$.

The above cases involve the main operations that we use for bit vectors: setting the least significant bit (case 1), shifting left by one position (case 3), and computing the disjunction of some of the most significant bits (case 4).

The way bit vectors are used (setting the lowest-order bit, shifting, and reading high-order bits) is similar to how queues and sliding windows are used for runtime

verification with metric temporal logic (MTL) [67, 211, 38, 232, 233]. We note that MTL involves constructs that specify time durations with intervals of the form $[m, n]$, which are akin to the bounded repetition construct $\{m, n\}$ of regexes. This explains the similarity in the implementation.

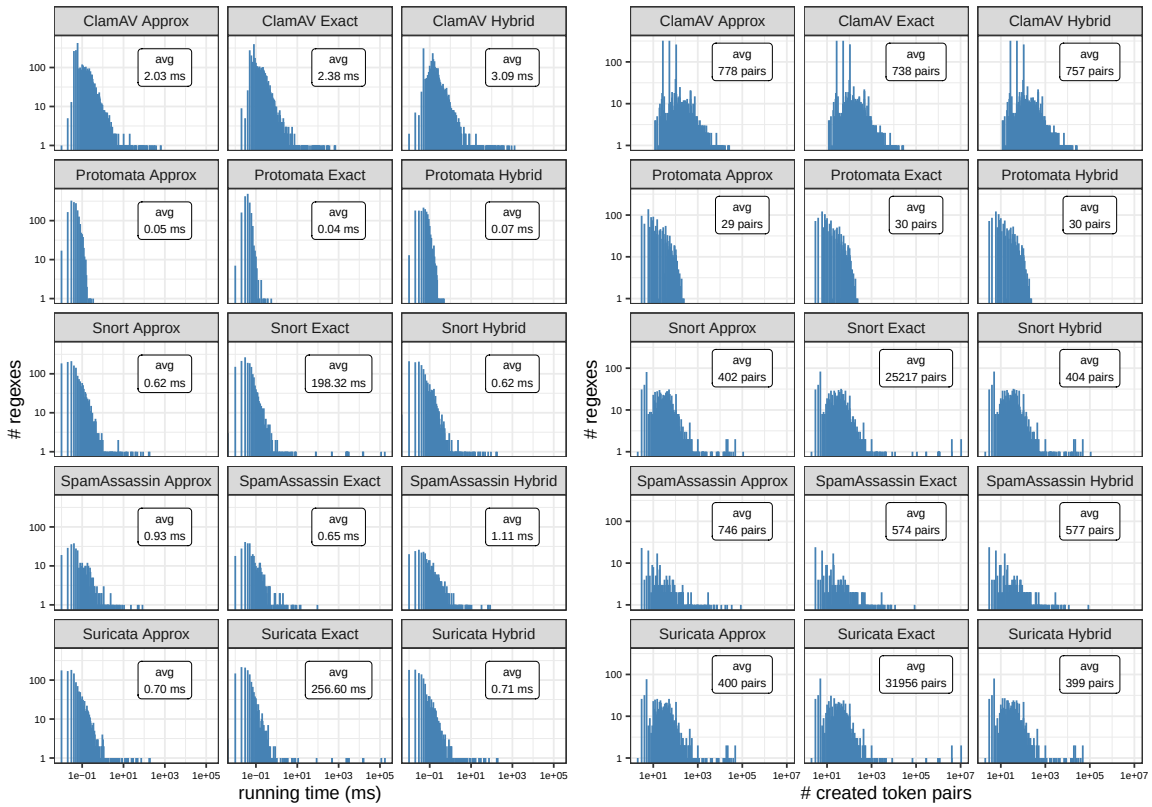
4.5 Experimental Evaluation of Static Analysis

We have implemented a Java program[§] that statically analyzes regexes to determine if they are counter-(un)ambiguous. We will call this program the *counter-ambiguity checker*. The implementation includes both the exact and the over-approximate analyses. As the approximate analysis may be unable to verify the counter-ambiguity of some instances, our checker implements a ***hybrid analysis***. First, it checks the counter-(un)ambiguity of each instance of bounded repetition in the regex using the over-approximate analysis. If it finds a potentially counter-ambiguous instance, then it halts the over-approximate analysis and uses the exact algorithm to check the regex. Otherwise, it determines that the regex is counter-unambiguous.

The checker not only determines if a regex is counter-ambiguous but also provides a *counter-ambiguity witness*, which is a string over the alphabet. If the NCA is executed on the witness, then at least two tokens with different counter valuations will end up on some state of the NCA. The checker supports the analysis of counter-ambiguity for each instance of bounded repetition inside a regex. For example, given a regex $\sigma_1\{m\}\Sigma^*\sigma_2\{n\}$, it can check the first instance (i.e., $\{m\}$), which is counter-unambiguous, and the second instance (i.e., $\{n\}$), which is counter-ambiguous.

We evaluate the performance of our counter-ambiguity checker using five benchmarks, which contain regexes collected from real applications. These benchmarks

[§]Link to the Java checker: `https://ohyoukillkenny.github.io/source/regexchecker.html`



(a) running time (b) # of created token pairs

Figure 4.2 : The (a) running time and the (b) # of created token pairs of static analysis for regexes. Exact means exact analysis, Approx means approximate analysis, and Hybrid means hybrid analysis. E.g., "ClamAV Exact" means the exact analysis in the ClamAV benchmark.

are: (1) the **Snort** [133] and (2) **Suricata** benchmarks [134] that contain patterns for network traffic, (3) the **Protomata** benchmark that includes protein motifs from the PROSITE database [135, 65], (4) the **ClamAV** benchmark [234] that contains patterns that indicate the presence of viruses, and (5) the **SpamAssassin** benchmark [229] that includes patterns for detecting spam email.

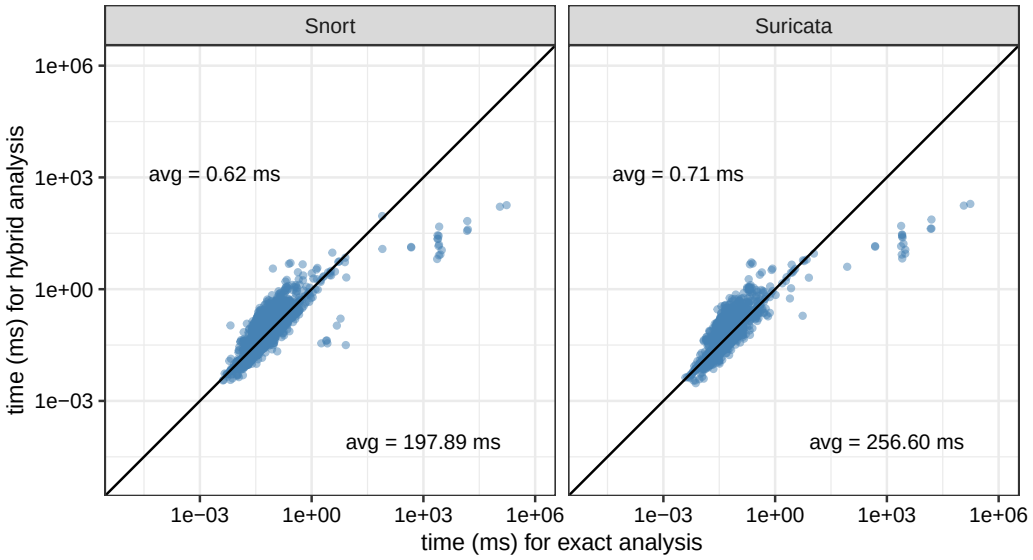


Figure 4.3 : Running time (ms) comparison of exact and hybrid analyses on the Snort and Suricata benchmarks.

Experimental Setup. The experiments were executed in Ubuntu 20.04 on a desktop computer equipped with an Intel Xeon(R) E3-1241 v3 CPU (4 cores) with 16 GB of memory (DDR3 at 1600 MHz). We used OpenJDK 17 and set the maximum heap size to 4 GB. For each regex, we executed 20 trials and selected the mean runtime as the value used in the reported results (excluding the first 10 "warm-up" trials).

4.5.1 Performance: Running Time

We evaluate the performance of the static analysis over regexes that have non-nested instances of constrained repetition. Figure 4.2(a) shows the distribution of the running time of the static analysis. The results are shown in 15 plots, which are organized in a 5×3 grid. There are 5 rows, one for each benchmark: ClamAV, Protomata, Snort, SpamAssassin, and Suricata. There are 3 columns, one for each variant of the static

analyzer: approximate, exact, and hybrid. The horizontal axis of each plot represents the time (in milliseconds) that our checker uses for analyzing regexes, and the vertical axis represents the number of regexes that need a certain amount of time for a variant of static analysis.

In the Snort and Suricata benchmarks, the checker takes more than 1 second to perform the exact analysis for several counter-unambiguous regexes. See the right outliers in the plots labeled "Snort Exact" and "Suricata Exact" in Figure 4.2(a). This information is seen more prominently in Figure 4.3, where the exact and hybrid analyses are compared on the Snort and Suricata benchmarks. The points with horizontal coordinate >1000 (msec) are noteworthy. They are substantially below the diagonal, which means that the hybrid analysis offers significant improvement in terms of running time. Some of these regexes are of the form $\Sigma^*(\bar{\sigma}_1\sigma_1\{m\} + \bar{\sigma}_2\sigma_2\{n\} + \cdots)$, where $m, n, \ldots$ are large numbers. When performing exact analysis on these regexes, the checker needs to explore a large number of token pairs, which makes the analysis time-consuming. However, as discussed in Example 4.4.2, the over-approximate analysis can greatly reduce the cost of the computation. We observe that the over-approximate analysis reduces the running time of expensive regexes by over 100 times in both the Snort and Suricata benchmarks. Moreover, as these regexes are counter-unambiguous, the result of their over-approximate analysis is accurate. This explains why the hybrid analysis also reduces the running time of these challenging regexes.

4.5.2 Performance: Memory Footprint

The checker analyzes the counter-ambiguity of a regex by exploring token pairs in a product transition system. These token pairs are created on the fly, as the transition

system is being explored. We estimate the memory footprint of the static analysis by measuring the number of token pairs that the checker creates. Figure 4.2(b) shows the results for five benchmarks and three different variants of the static analysis. Similarly to the case of running time, the over-approximate analysis greatly reduces the worst-case cost of analyzing several counter-unambiguous regexes in the Snort and Suricata benchmarks.

4.6 Hardware Design for Efficiently Executing NCAs

In collaboration with researchers from SIMS Lab[¶] at Rice University, we augmented a state-of-the-art in-memory NFA acceleration architecture called CAMA [53] with counter and bit vector modules. In this section, we will present our hardware design for efficiently executing NCAs.

Existing in-memory automata accelerators adopt a two-phase architecture: a state matching phase that finds the current active states, and a state transition phase that calculates the available states in the next cycle. AP-style accelerators, such as AP [54], CA [225], and eAP[‖] [235], perform state matching by reading from read-access memories (RAMs) that store bit vector representations of states in memory columns. Each column in the RAM represents one state, which is called a State Transition Element (STE). Using 8-bit symbols as an example, each RAM entry is 256-bit and the i-th position has value 1 iff the symbol i is associated with the state[**]. Additionally, the connections between states are programmed into a switch network where existing

[¶] `https://vlsi.rice.edu/`

[‖] eAP stands for embedded Automata Processor.

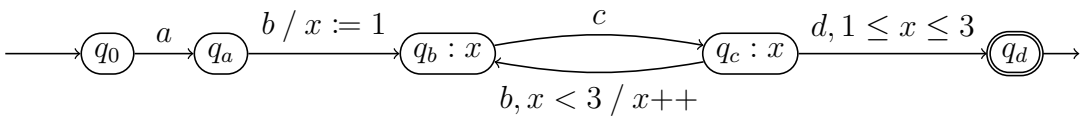
[**] Recall from Section4.3 that we consider homogeneous automata, which means that all transitions leading to a state q are labeled with the same predicate σ over the alphabet. The RAM entry is a representation of the predicate σ.

state transitions are realized as physical connections.

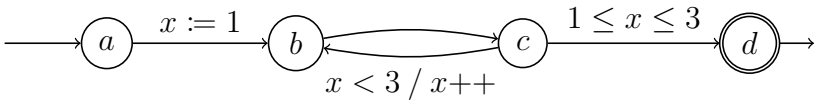
Each processing cycle begins in the state matching phase, where an input symbol is encoded as a one-hot representation[††] and used as the address to read from the state matching memory. The columns that read out '1's indicate successful matches between the input symbol and the STEs. With a logical AND operation between the available states reported from the last cycle and the matched states reported by the memory in the current cycle, matching results of the active states in the current cycle are determined. Next, in the state transition phase, the current active states pass through the programmed switch network to create the next vector which stores available states for the next cycle.

However, AP-style accelerators severely under-utilize the state matching memories in realistic NFAs across common benchmarks, because this approach is optimal only for the worst case of purely random NFAs. Impala [226] and CAMA [53] made critical improvements by proposing special encoding schemes to reduce the state matching memory requirements. CAMA further employs specialized content-addressable memories (CAM) to perform state matching with lower energy and memory footprints than all other designs using RAM. Moreover, CAMA optimizes a reduced-crossbar switch network that was first proposed by eAP, which largely reduces the area and energy costs of state transitions. Compared with prior NFA in-memory architectures, CAMA achieves leading throughput, energy, and area efficiency. The throughput of CAMA is 2.14GBps, which is 1.18 times better than CA, 9.5 times better than FPGA-based Grapefruit [73], and 2-4 orders better than CPU/GPU solutions. The energy efficiency of CAMA is 4.91nJ/Byte, which is over 10 times

[††]The one-hot representation of an 8-bit symbol i consists of $2^8 = 256$ bits, where the i-th bit has value 1 and the others are 0.



(a) Glushkov NCA for regex $a(bc)\{1,3\}d$



(b) NCA with STEs for regex $a(bc)\{1,3\}d$

Figure 4.4 : The (a) Glushkov NCA for regex $a(bc)\{1,3\}d$ and the (b) corresponding NCA with STEs.

better than most efficient alternatives, i.e. Grapefruit (FPGA) and AP. We use the latest memory- and energy-efficient CAMA architecture as the baseline and augment it with our proposed counter and bit vector modules.

Figure 4.4 shows the Glushkov NCA for the counter-unambiguous regex $a(bc)\{1,3\}c$. The Glushkov construction ensures that the NCA is homogeneous (all transitions entering a state are labeled with the same predicate over the alphabet). This property allows us to convert the NCA to a hardware-friendly representation by omitting the initial state and pushing the predicates from the edges to the states, thus transforming NCA states into STEs. For example, we push the predicate a into state q_a so that in Figure 4.4(b) we have a state labeled with the predicate a, which becomes an STE that is activated to fire signals only when the input satisfies the predicate a.

The original CAMA design, as shown in Figure 4.5(a), only supports NCAs by fully unfolding bounded repetitions. In our augmented CAMA, two types of hardware modules, counters and bit vectors, are added to accelerate the execution of NCAs. As shown in Figure 4.5(b), both modules take input from STEs related to counting and produce output signals to the switch network. Counters are inserted to support counter-unambiguous repetitions, while bit vectors are reserved for counter-ambiguous

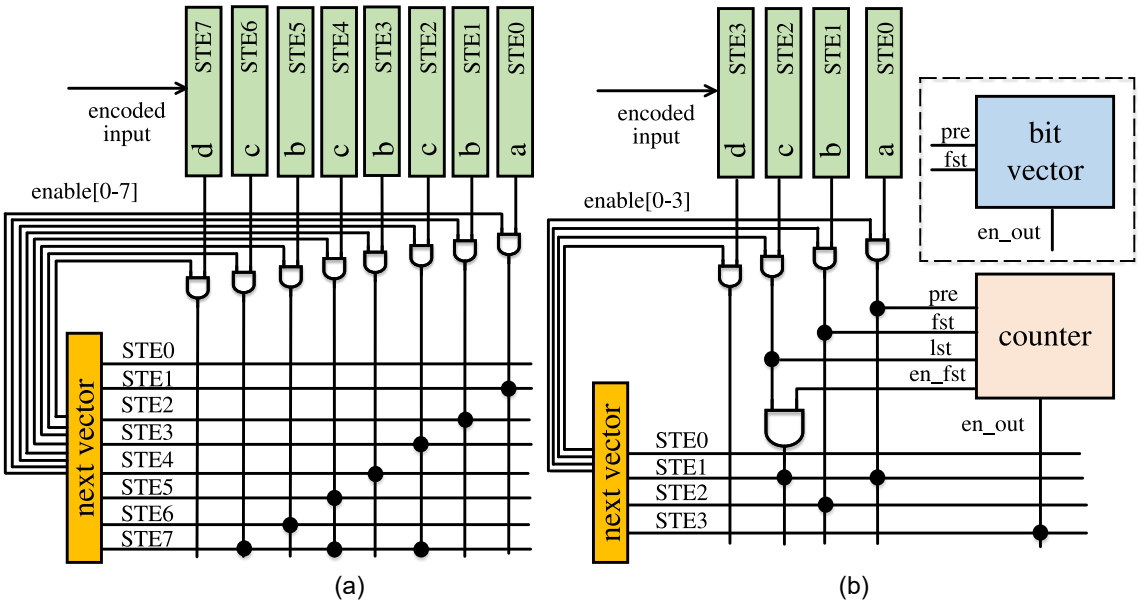


Figure 4.5 : (a) shows the CAMA design with the unfolding of regexes. (b) shows our augmented design with the counter or the bit vector.

repetitions. Compared to CAMA, the additional counters and bit vectors retain all necessary processing information while avoiding the cost of unfolding (which results in additional STEs). In Section 4.7, we will further explain the design and the input/output ports of the counter and bit vector modules.

It is worth mentioning that our proposed counters and bit vectors are not only suitable for the CAMA architecture. Other in-memory automata architectures, like CA, can also be augmented for NCAs with minor hardware design changes. Specifically, these changes are: (1) counters and bit vectors need to be allowed to connect to elements that represent states, and (2) the routing network needs to be extended to store the transitions from counters and bit vectors.

The initial motivation for our hardware design came from the observation that several instances of bounded repetition require significantly less memory than what is

suggested by a naïve unfolding. This led to the formalization of counter-(un)ambiguity in NCAs and the corresponding static analysis. For the counter-unambiguous case, it suffices to use simple counter modules that keep track of the number of repetitions. For the counter-ambiguous case, the use of bit vectors is a very natural choice for a hardware representation of sets of tokens. These considerations led to the design of the counter and bit vector modules. Physical constraints imposed by the hardware call for minimizing the connections between STEs and the counting modules. For this reason, we have chosen to use bit vectors for counter-ambiguous repetitions of the form $\sigma\{m, n\}$ and use (partial) unfolding for other cases. The vast majority of counter-ambiguous repetitions in real-world benchmarks are of this form, so this approach offers efficiency (due to an optimized hardware implementation) without sacrificing generality (since the remaining cases can be handled at the level of the software/compiler).

4.7 Compilation from Regular Expressions to MNRL Files

To program the hardware, we provide a description of the automata in the MNRL language [228]. Our compiler takes a source regex and produces the MNRL file with the following steps: (1) First, the compiler parses the regex and simplifies it with certain rewrite rules, including the unfolding of repetitions with upper bound < 2 and the merging of character classes inside simple alternations (e.g., `[a]|[b]` is rewritten to `[ab]`). (2) Then, the compiler performs the static analysis (as described in Section 4.4) and annotates the regex with the counter-(un)ambiguity result for each occurrence of repetition. (3) Finally, the compiler generates the MNRL file using these annotations, distinguishing cases where a counter suffices (counter-unambiguous) from cases where a bit vector is necessary (counter-ambiguous).

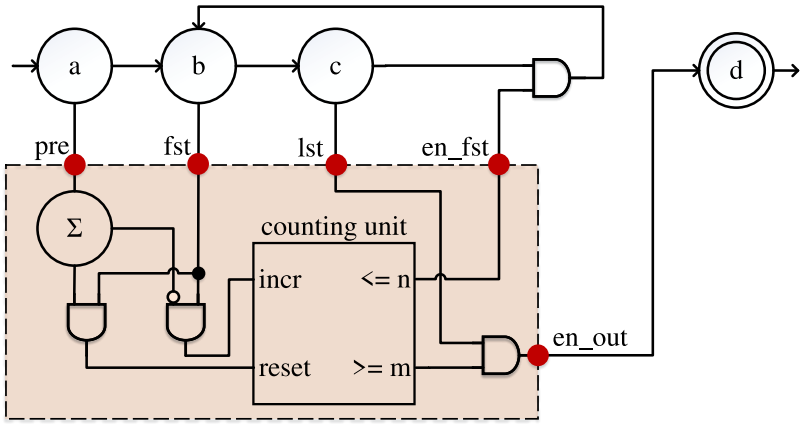


Figure 4.6 : Use of counter module to implement $a(bc)\{m,n\}d$.

MNRL provides an element called `upCounter` for representing simple counters [54, 228]. However, there is no distinction between counter-ambiguous and counter-unambiguous repetition. We have therefore extended the MNRL format by adding syntax for counters and bit vectors.

Figure 4.6 presents an abstraction of the *counter module* (enclosed by a dashed line) by showing how it is used to implement the counter-unambiguous regex $a(bc)\{m,n\}d$ in hardware. A counter has three incoming ports `pre`, `fst`, and `lst`, and two outgoing ports `en_fst` and `en_out`, where ports are labeled with red dots in Figure 4.6. The input port `pre` (i.e., pre-counting) is connected to the STE (labeled with a) located right before the repetition, `fst` (i.e., first) is connected to the first STE (labeled with b) in the repetition, and `lst` (i.e., last) is linked to the last STE (labeled with c) in the repetition. The output port `en_out` (i.e., enable output STE) activates the STE (labeled with d) located right after the repetition, and `en_fst` (i.e., enable first STE) activates the first STE (labeled with b) in the repetition. The counter module consists of a synchronous counting unit using D flip-flop and two digital comparators. The module is designed to meet four constraints: (1) The counter value is reset to 0 when

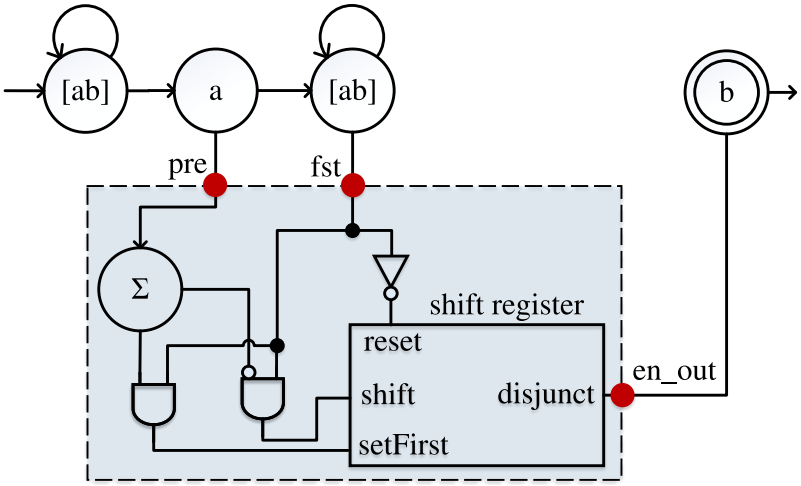


Figure 4.7 : Use of bit vector to implement $[ab]^*a[ab]\{m,n\}b$.

`pre` was active in the previous cycle and `fst` is currently active. This corresponds to the initialization of the repetition. (2) The counter value is incremented by 1 when `fst` is active but `pre` was not active in the previous cycle. This corresponds to one complete cycle. (3) `en_out` fires if `lst` is active and the counter value is within the expected range (i.e., $[m, n]$). (4) `en_fst` fires if `lst` is active and the counter value is $\leq n$.

Figure 4.7 presents an abstraction of the *bit vector module* by showing how the regex $[ab]^*a[ab]\{m,n\}b$ is implemented in hardware. The core component of the bit vector is a serial-in-parallel-out shift register. It supports four primary operations: (1) `reset`, which resets all bits in the vector to 0, (2) `setFirst`, which sets the first bit of the vector to 1, (3) `shift`, which shifts the vector by one bit, and (4) `disjunct`, which computes the disjunction of a sub-array of bits from index m to n (if one of the bits in the sub-array is 1, the output signal fires).

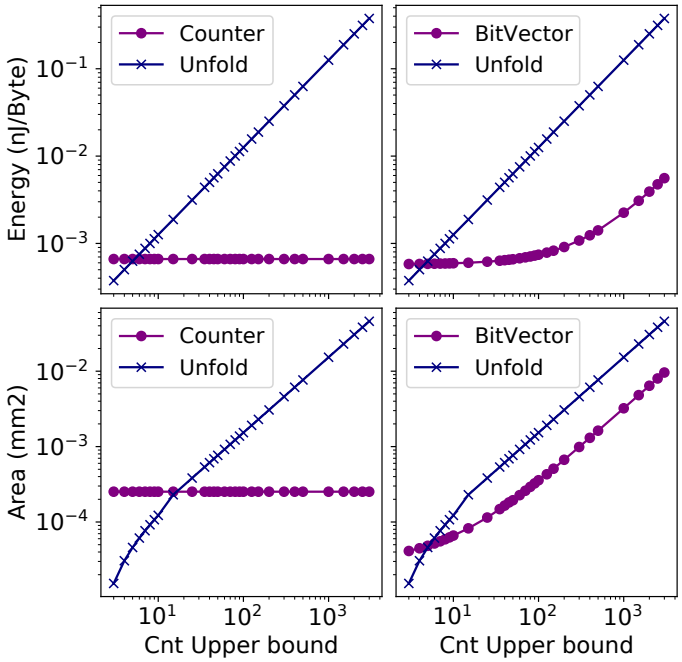


Figure 4.8 : Energy (upper two figures) and area (bottom two) trade-off of unfolding vs using counter (left two figures) and bit vector (right two), where axis is log-scaled.

4.8 Evaluation of Hardware Performance

We modified the open-source simulator VASim [70] to simulate the hardware performance of our augmented CAMA. We include 17-bit counters for supporting unambiguous counting, and 2000-bit vectors for supporting ambiguous counting.

4.8.1 Micro-benchmarks

Figure 4.8 shows the trade-off of unfolding vs. using counter and bit vector modules. In the left two sub-figures, we consider regexes $a\{n\}$ with different values of n. These regexes are counter-unambiguous – the hardware implementation only needs a single counter module to perform the matching, while unfolding creates n STEs. The upper-left (resp., bottom-left) sub-figure shows the energy (resp., area) cost of using a counter

module compared with unfolding, where we always use a 17-bit counter module to represent counter values regardless of their different repetition bounds. In the right two sub-figures, we consider regexes $\Sigma^* a\{n\}$. These regexes are counter-ambiguous, so the hardware needs to use a bit vector to perform matching, while unfolding creates n STEs. In this comparison, we set the length of the bit vector to be equal to n for each data point (this implies that bits are wasted). The upper-right (resp., bottom-right) sub-figure shows the energy (resp., area) cost of using a bit vector compared with unfolding. From the results shown in Figure 4.8, we observe that using a counter/bit vector provides better performance compared to unfolding even for repetitions with small upper bounds. It consistently reduces energy usage by orders of magnitude and areas by large margins.

4.8.2 Real-world Benchmarks

We use the same benchmarks as described in Section 4.5 (except for ClamAV). Figure 4.9 shows the number of MNRL nodes (which is linear in the number of STEs) for different unfolding thresholds. For each benchmark and each point in the corresponding curve, the horizontal axis shows the unfolding threshold k and the vertical axis shows the number of MNRL nodes that are obtained from compiling the entire benchmark after bounded repetitions up to k have been unfolded. The rightmost point on each benchmark curve shows the unfolding threshold that results in full unfolding for all regexes of the benchmark and the resulting number of MNRL nodes.

We have simulated the area and the energy consumption of our augmented CAMA by feeding compiled MNRL files with different unfolding thresholds to the modified VASim using Protomata, SpamAssassin, Snort, and Suricata benchmarks. Figure 4.10

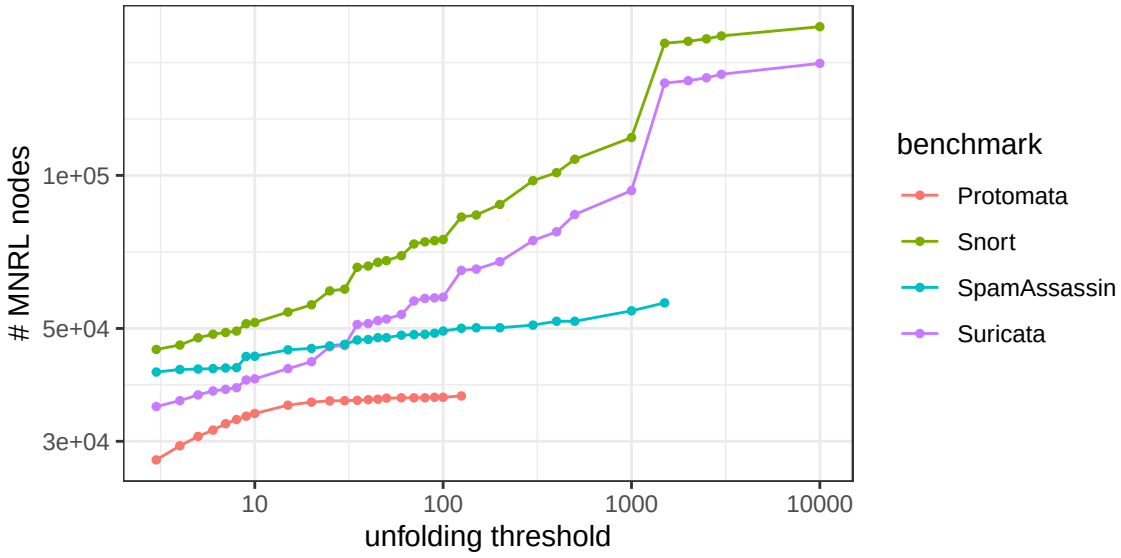


Figure 4.9 : Total number of MNRL nodes with different unfolding thresholds (both axes are log-scaled).

shows the per-input-byte energy consumption and the total area cost of the augmented CAMA. The results show up to 76% energy reduction and 58% area reduction in benchmarks with an abundance of instances of bounded repetition with large upper bounds (i.e., Snort and Suricata). In benchmarks that generally include bounded repetitions with small upper bounds (i.e., Protomata and SpamAssassin), the augmented CAMA hardware still outperforms pure CAMA with little to no overhead. We observe that for the Protomata and SpamAssassin benchmarks, our hardware implementation provides less energy and area reduction compared with Snort and Suricata. This is because, in general, the regexes in Protomata and SpamAssassin have small repetition upper bounds. The wasted area in Figure 4.10 corresponds to unused bits in the bit vector modules.

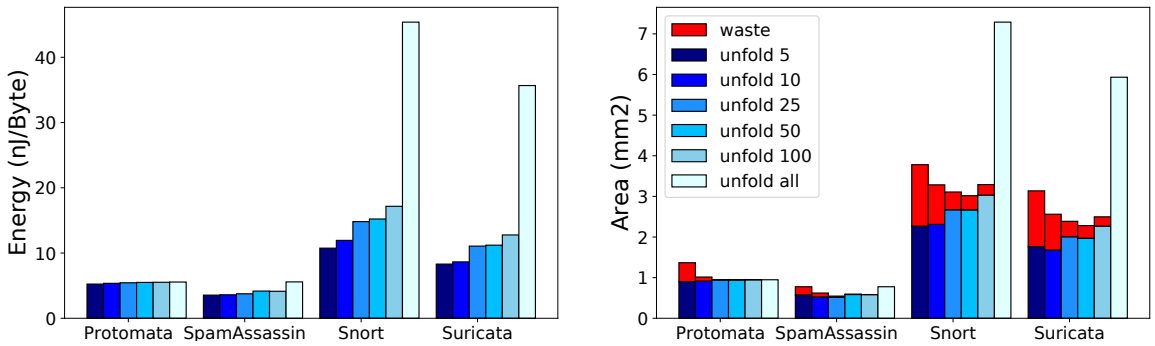


Figure 4.10 : Per-input-byte energy consumption (left) and total area cost (right) of the augmented CAMA hardware.

4.9 Chapter Summary

We have investigated hardware acceleration for regular pattern matching, where the patterns are specified by regexes with an extended syntax that involves bounded repetitions of the form $r\{m, n\}$. We have developed a design that integrates counter and bit vector modules into an in-memory NFA-based hardware architecture. This design is inspired from the theoretical model of nondeterministic counter automata (NCAs) and the observation that some instances of bounded repetitions require only a small amount of memory. We formalize this idea using the notion of counter-unambiguity. We have implemented a regex-to-hardware compiler that performs a static analysis for counter-(un)ambiguity over a regex and then creates a representation of an automaton with counters and bit vectors that can be deployed on the hardware. Our experiments show that using counters and bit vectors outperforms unfolding solutions by orders of magnitude. Moreover, in experiments with real-world workloads, we have observed that our design can provide up to 76% energy reduction and 58% area reduction in comparison to CAMA, a state-of-the-art in-memory NFA processor.

Chapter 5

Semantics-preserving Parallel Stream Processing

5.1 Motivation

Applications that handle massive streaming data have scalability requirements and would benefit from a multicore implementation of the streaming language/engine. For example, in the application related to high-frequency stock trading, billions of quotes and transactions occur per day [80]. The non-trivial analysis applied to these data, which have such high volume and velocity, requires a multicore implementation that meets the throughput requirement. Additionally, applications that deal with streaming data typically have strict correctness requirements regarding the in-order processing of input items. For various workloads such as the analysis of price patterns in the stock market [26] and the detection of arterial blood pressure pulses over sensor measurements [81], a *safe* parallel implementation, as emphasized in [1], should process items strictly based on their input order and produce the same result as the sequential implementation.

In this dissertation, we focus on the parallel processing of streaming data on a multicore CPU. The communication between different threads in this scenario is more reliable than in a distributed system as messages/data items are delivered with lower latency and are less likely to be lost compared to network communication. To parallelize the processing of data items, programmers typically describe the computation as a dataflow graph, where each node corresponds to a stage of the overall computation.

During compilation and deployment, this dataflow graph is mapped to physical cores and processes to achieve parallelism, which is used to accelerate the overall streaming computation as multiple nodes can perform the computation simultaneously.

There is a variety of tools developed for exposing parallelism to stream processing based on the dataflow graph. However, they often fail to provide sufficient support for processing streaming data with the guarantee of safety in our context – the output of the parallel computation should be the same as the sequential computation. For example, systems for distributed stream processing [82, 22, 20, 83, 84, 236] such as Storm [20] cannot provide support for preserving the sequential order of outputs, and others such as Flink [83] utilize external APIs to reorder the timestamps of the input items. Synchronous dataflow (SDF) languages and models [85, 30, 86, 31, 55] of computation are useful for implementing the parallelism present in streaming computations in the embedded software domain. However, SDF languages typically assume fixed item rates, meaning that given an input item, a fixed number of output items will be generated in the computation. This property is used for efficient scheduling determined at compile-time, but the downside is that SDF languages cannot natively express computations that involve dynamic item rates, where given an input item, an arbitrary number of output items may be generated. A recent study from IBM [1] proposes a framework to ensure safe data parallelism even if the streaming computation has dynamic item rates. It provides multiple ordering strategies to preserve the sequential semantics by assigning a sequence number to each data item and reordering items by their sequence numbers. However, these strategies, in certain cases, are not sufficient to maintain the sequential semantics when the computation is implemented with non-linear task parallelism ("non-linear" indicates tasks cannot simply fit in a pipeline). Timely Dataflow [82, 237] is a powerful framework for efficiently implementing data

parallelism in streaming data processing. However, when the preservation of sequential semantics is required, the throughput speedup may not be optimal. This is because the timely dataflow engine needs to wait for the advancement of timestamps in the processing of each data item, which can limit the potential for parallelism and hinder overall throughput improvements.

5.2 Contributions

We proposed a novel ordering strategy for data items that diverges from traditional methods by utilizing *signature*s. A signature is as a series of sequence numbers that unambiguously indicates the processing order of the data. We have developed a general framework that is built on this signature-based approach and is able to preserve the sequential semantics. This framework is able to handle data items with duplicate timestamps, adapt to dynamic item rates of operations, and accommodate non-linear forms of task parallelism.

We have developed a lightweight Rust library called ParaStream, which safely parallelizes the processing of data streams. Rust was chosen as it provides a user-friendly environment for safe and efficient concurrent programming, known as "fearless concurrency" [238]. ParaStream includes a comprehensive set of operators for describing streaming computations performed by a node in the dataflow graph, such as primitives for transforming, filtering, and aggregating streams, a group-by combinator for key-based partitioning and independent computation over disjoint sub-streams, and tumbling/sliding windowing combinators for computations that operate on finite spans of an unbounded data stream. Our algorithm for preserving sequential semantics is implemented on top of each node, allowing users to program their computations with safe parallelism without the need for low-level item reordering algorithms. We

have compared ParaStream to popular lightweight libraries for efficient processing of streaming data: StreamQL [87], RxJava [205], Reactor [128], and Timely Dataflow [239] in single-threaded execution. The results show that ParaStream is 1.2-6 times faster than StreamQL, 2–20 times faster than Timely Dataflow, 3-25 times faster than RxJava, and 8-50 times faster than Reactor. Additionally, we have evaluated the throughput scalability of ParaStream and observed that ParaStream offers superior scalability compared to RxJava, Reactor, and Timely Dataflow. Furthermore, we have observed near-linear throughput speedups of ParaStream in several real-world benchmarks as the degree of parallelism increases.

As the summary of our contributions:

1. We present a novel framework for the parallel processing of streaming data that preserves sequential semantics, where our framework handles challenges not fully addressed in prior works.

2. We provide the ParaStream library, which offers high-level operators for programming streaming computations, and supports safe parallelism with the implementation of our semantics-preserving algorithms.

3. We provide a high-throughput library for supporting parallel streaming computations. The benchmarking results show that ParaStream consistently provides higher throughput than state-of-the-art tools in single-threaded settings and substantial speedups with increasing degrees of parallelism, despite the additional overhead of preserving sequential semantics.

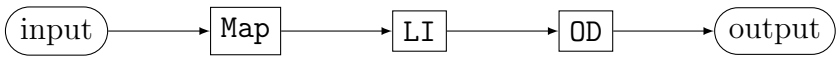


Figure 5.1 : The three-stage pipeline of the outlier detection algorithm.

5.3 Challenges in Preserving Sequential Semantics

In this section, we will use concrete examples to describe parallel patterns commonly used in real-world applications and discuss the challenges of implementing these patterns while preserving sequential semantics. We will focus on the processing of a stream of sensor measurements collected from different sensors, which is a time series with non-decreasing timestamps. It is important to note that multiple data items may have the same timestamps as they are detected by different sensors, and the time series may have missing data points due to sensor failures.

We will demonstrate the use of a three-step outlier detection algorithm for measurements collected by each sensor. This algorithm will first (1) detect measurements that match a specific pattern (e.g., peak detection), then (2) update the temporal summary of measurements (e.g., 1-hour average), and (3) compare the values of measurements detected in the first step to the temporal summary obtained in the second step to identify outliers. The stream processing pipeline, as shown in Figure 5.1, includes three stages: (1) the `Map` stage deserializes incoming measurements from sensors and retains only the scalar value, sensor identifier, and timestamp (discarding other metadata), (2) the `LI` stage performs linear interpolation to fill in missing data points for measurements collected by each sensor, and (3) the `OD` stage detects outliers from measurements from every sensor.

Such a pipeline can be conveniently implemented as a dataflow graph by providing the code for each node and using FIFO channels to pass data items between these

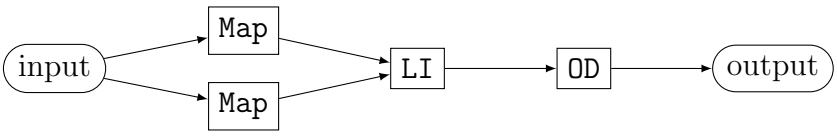


Figure 5.2 : Dataflow graph with parallel transformation I.

nodes. The implementation described above exposes pipeline parallelism and therefore suggests a multi-process execution, where each stage of the pipeline runs as an independent process.

5.3.1 Parallel Transformations and Corresponding Challenges

To further improve the performance of the streaming computation, we apply three parallel transformations to this implementation. These parallel transformations introduce additional nodes to the dataflow graph, providing more opportunities for parallelization to enhance the speed of computation. For each parallel transformation, we will also discuss the challenges related to preserving the sequential semantics of the original pipeline.

Parallel Transformation I

When sensors produce measurements at a very high rate, the deserialization stage `Map`, which is computationally expensive, becomes a bottleneck. To alleviate this, we can create several parallel instances of `Map` and enhance the throughput. The implementation splits and balances the input stream across multiple `Map` instances, and it then merges the output streams of the `Map` instances. The merged stream is then sent to the `LI` stage for further processing. Figure 5.2 shows the dataflow graph after this parallel transformation is performed.

The linear interpolation stage `LI` relies on receiving data items in increasing order

of timestamps. However, if an algorithm directly merges the output streams of the multiple `Map` instances, it can introduce arbitrary interleaving that may violate this precondition. This can lead to non-determinism in the system, making the outputs unpredictable and non-reproducible.

Parallel Transformation II

The computationally expensive processing applied to measurements collected from each sensor (i.e., `LI` + `OD`) also causes a bottleneck. To address this, we split the output from `Map` into multiple sub-streams, where each sub-stream contains measurements from a unique sensor. We then create multiple parallel instances of the `LI` and `OD` nodes, with each pair of `LI` and `OD` processing a set of these sub-streams. Figure 5.3 shows the dataflow graph after performing the second parallel transformation.

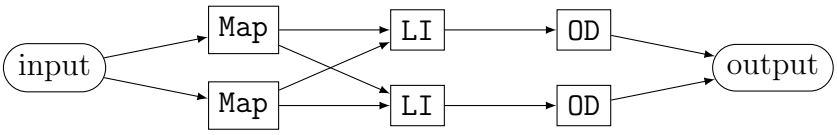


Figure 5.3 : Dataflow graph with parallel transformation II.

Similar to the first parallel transformation, the issue with the second parallel transformation is that if we directly merge the results produced by multiple `OD` instances, the interleaved output stream can contain out-of-order data items (i.e., with decreasing timestamps).

Parallel Transformation III

Finally, we can enhance the throughput of the computation performed in each `OD` node by introducing non-linear task parallelism to the execution. We split the

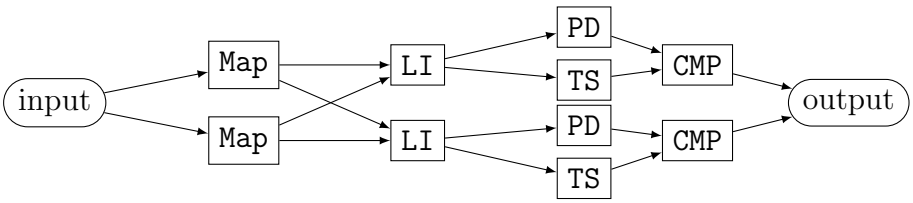


Figure 5.4 : Dataflow graph with parallel transformation III.

computation performed on each OD node into three tasks as previously discussed: pattern detection (performed on PD), temporal summarization (TS), and comparison (CMP). The implementation below splits each OD node into three nodes, where PD and TS are executed in parallel and their results are merged by CMP. Figure 5.4 shows the corresponding dataflow graph.

We use CMP to collect the results produced by PD and TS. If the computation of PD and TS is performed on different threads, CMP, without a reordering approach, may fail to process the results from PD and TS based on their order of generation, thereby breaking the sequential semantics.

5.3.2 Limitations in Prior Proposals

Functional reactive programming (FRP) related libraries like Rx [127, 39], Reactor [128], and Akka Streams [129] are widely used to process the streaming data, particularly for event-driven and interactive applications such as GUIs. These libraries address concurrency and support parallelism by describing the computation performed on different nodes as the transformation of reactives (signals and events) [40, 124, 42, 125] and using a scheduler to perform the transformation using different threads. However, when using these libraries, if we assign multiple threads to perform a single step of the transformation simultaneously and merge the results from these threads (e.g., the first parallel transformation), we cannot preserve the order of

the outputs as the same as the sequential program.

The synchronous dataflow (SDF) languages and models of computation are useful for expressing parallelism present in streaming computations [30, 86, 31, 55]. In particular, StreamIt [55] provides a general framework for streaming signal processing with efficient execution on multicore architectures. StreamIt assumes fixed item rates (i.e., number of output items generated per input item) – a property used by SDF for efficient scheduling determined at compile-time. Therefore, while it can preserve the sequential semantics for the first parallel transformation since the item rate of `MAP` is always 1, StreamIt cannot directly be used to program parallel transformations such as the second transformation, where `LI` has dynamic item rates.

There are language-based approaches that enforce semantics-preserving parallelization for processing data streams [200, 1, 202, 199, 201]. In particular, [1] has presented a runtime system that can preserve the sequential semantics in the presence of operators with dynamic item rates. This system generates sequence numbers and attaches them to data items to later recover their order (if they get out of order). However, such an order-restoration scheme with sequence numbers is not sufficient to preserve the semantics in the presence of deep nesting of order-dependent parallel computations. For example, in the third parallel transformation, let us consider the input items of `LI` are assigned with unique sequence numbers, i.e., in Figure 5.5, a_1, b_2, and c_3 respectively have sequence numbers 1, 2, and 3. If `LI` produces multiple outputs for an input item (e.g., f and g are generated by b), it will, based on the order-restoration schemes described in [1], assign the same sequence number (e.g., 2) to these outputs. After the output of `LI` is processed by `PD` and `TS`, the sequence number shared by multiple items (e.g., 2) may be propagated and sent to `CMP`. In this case, `CMP` can process items in a non-deterministic order (e.g., ...m_2 l_2 j_2 i_2...,

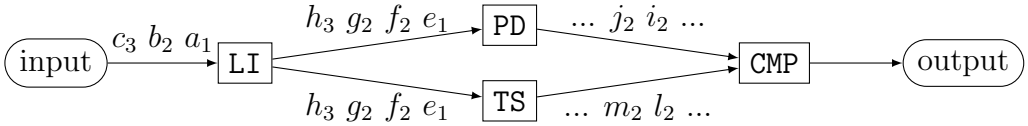


Figure 5.5 : The use of sequence number to preserve the sequential semantics.

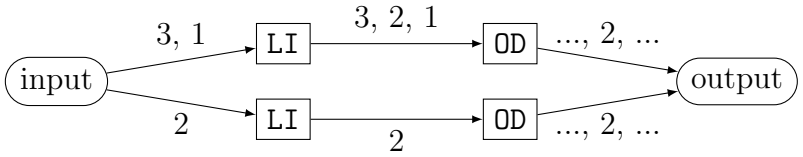


Figure 5.6 : The use of timestamp to preserve the sequential semantics.

...$j_2\ m_2\ l_2\ i_2$..., etc.), thus generating a different result compared to the sequential program.

Trill [106] is a high-performance streaming library that employs a batched-columnar data representation and dynamic compilation. Trill supports parallel stream processing by providing streaming generalizations of the classic MapReduce operations with temporal support, and it does not natively support parallel patterns which involve task parallelism such as the third parallel transformation. Trill preserves the semantics by reordering data items using their timestamps. However, this schema is not sufficient when multiple items carry the same timestamp. For example, in the second parallel transformation, let us consider two `LI` instances that receive data items with integer timestamps "$1, 3$" and "2" respectively (see Figure 5.6). Suppose `LI` fills in the missing data items for each time unit, the `LI` instance on the top will generate output data items with timestamps "$1, 2, 3$", where the interpolated data item shares the same timestamp with data item sent to the `LI` instance on the bottom. After `OD` propagates the output of `LI` instances, the sink of the pipeline may encounter two data items with the same timestamp (e.g., 2), which introduces non-determinism.

Based on the previous discussion, we have identified several challenges to be addressed for the preservation of sequential semantics in the processing of streaming data. These challenges include handling computation that generates data items that share the same timestamp, operations with dynamic item rates, and non-linear task parallelism. To the best of our knowledge, no prior work has fully addressed these challenges. As a result, users are left with two unsatisfying solutions: (1) they can either accept these shortcomings and execute their computations with restricted parallelism, which may introduce a throughput bottleneck, or (2) they can manually preserve the semantics by implementing low-level item reordering algorithms for each node in the dataflow graph, which is arduous and error-prone.

5.4 Solution from ParaStream

We propose a framework, called ParaStream, to support the parallel processing of streaming data while preserving the sequential semantics. Our solution is based on the use of ***signature***s. A signature is a sequence of natural numbers that indicates the expected processing order of each data item. By comparing the signature of each data item in a lexicographic order, we can ensure data items always exit nodes in the dataflow graph in the same order as they would without parallelization, thus preserving the sequential semantics. In this section, we will first present the construction of signatures for each data item. Then, we will show how we use signatures to handle all aforementioned challenges, such as data with the same timestamps, computations that have dynamic item rates, and the occurrence of non-linear task parallelism.

ParaStream uses a directed acyclic graph (DAG) to describe a streaming computation that is performed on multiple threads. Each node of the DAG, for each input item, monotonically generates outputs, where the monotonicity captures a key

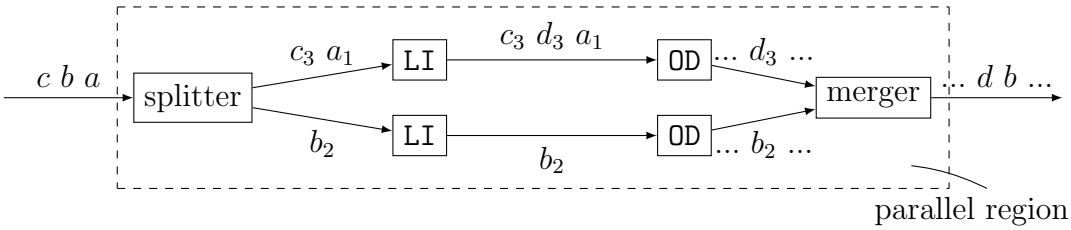


Figure 5.7 : Dataflow graph with the use of signatures.

feature of the streaming computation: an output item cannot be retracted after it has been emitted. Each edge of the DAG is a concurrent FIFO channel that transmits data from one node to another. A ParaStream DAG is constructed by three kinds of nodes: *splitter*, *worker*, and *merger*, where a splitter receives data items from a single input channel and sends output items into multiple output channels, a merger collects input items from multiple input channels and provides outputs using a single output channel, and a worker gets inputs from a single input channel and delivers output to a single output channel. Additionally, each merger corresponds to a splitter such that the merger collects the sub-streams originating from the partition performed by the splitter and merges them as its output stream. Figure 5.7 shows a DAG for PT2, where a splitter splits an input stream "a, b, c" and sends the sub-streams to workers that perform linear interpolation and outlier detection, and a merger collects the results from `OD` workers. We call the subgraph of a DAG that contains a splitter, its corresponding merger, and all involved workers a *parallel region*.

The splitter propagates a data item that enters a parallel region using the following algorithm: it counts the number of emitted outputs as n, and for each input item with signature u, it assigns $u \cdot n$ as the signature of the output generated by this input item. Inside a parallel region, each worker assigns the same signature of each input item to the outputs that are generated by it. Finally, the merger always first processes

the input with the smallest signature received from all of its input channels. Then, for each output generated by this input, the merger assigns a new signature to it by removing the rightmost integer from the signature of the input.

ParaStream reorders the data based on their signatures, which indicate the occurrence order of each data item. Therefore, in cases where several data items have the *same timestamp*, ParaStream can determine which item to process first since their order of occurrence is unique. For example, in the previous figure, after "a, b, c" enter the parallel region, a splitter assigns a signature to each input item (e.g., the signature of a_1 is 1). `LI` and `OD` workers in the parallel region copy the signature of each input and assign it to the output generated by this input. For example, for the `LI` node on the top, d is a data item generated during the linear interpolation. This item, generated by c_3, inherits the signature carried by c_3, i.e., 3. When the merger merges the results generated by `OD` workers, it can provide outputs in a deterministic order such that b is always emitted before d since $2 < 3$.

ParaStream preserves the sequential semantics even when the computation performed on a node in the DAG has a *dynamic item rate*, where an arbitrary number of outputs are generated by an input. If such a node is a worker or a merger, the order of outputs is preserved automatically by the FIFO implementation of the output channel since both worker and merger have only a single output channel. If such a node is a splitter, we distinguish the order of its output items by assigning different signatures to them. For example, for an input item carries signature u, if a splitter generates two output items, it will assign these output items signatures $u \cdot n$ and $u \cdot (n + 1)$ respectively, n is the current count of output items, based on their order of generation.

Furthermore, we can also use signatures to preserve the semantics in terms of the execution of *non-linear task parallelism*. For example, in PT3 (see Figure 5.8),

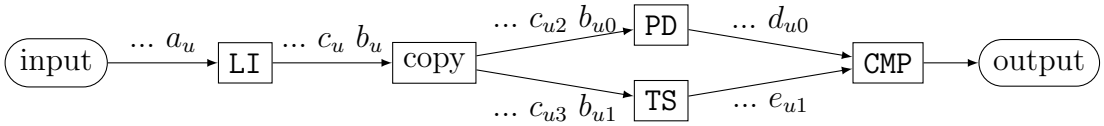


Figure 5.8 : The use of signature for dataflow graph with non-linear task parallelism.

Table 5.1 : Parallel patterns supported with the preservation of sequential semantics in ParaStream, StreamIt, Rx, [1], Trill, and Timely Dataflow (✓/✣/✗ indicates whether the pattern is supported, conditionally supported, or not supported, ✓/✣/✗ indicates whether the sequential semantics is preserved, conditionally preserved, or not preserved).

	ParaStream	StreamIt	Rx	[1]	Trill	Timely
pipeline parallelism	✓/✓	✓/✓	✓/✓	✓/✓	✗/—	✓/✓
data parallelism	✓/✓	✣/✓	✓/✗	✓/✓	✓/✣	✓/✣
MapReduce	✓/✓	✗/—	✓/✗	✓/✓	✓/✣	✓/✣
non-linear task parallelism	✓/✓	✣/✓	✗/—	✓/✣	✗/—	✓/✣
mixed data/task parallelism	✓/✓	✣/✓	✗/—	✓/✣	✗/—	✓/✣

suppose `LI` generates two outputs (i.e., b and c) for an input item a. `LI`, as a worker, will directly assign the signature of the input to the output. Therefore, a, b, c carry the same signature (i.e., u). The splitter that copies the data item will extend the signature by a sequence number, and it will send items b_{u0} and c_{u2} (resp., b_{u1} and c_{u3}) to `PD` (resp., `TS`). We suppose `PD` (resp., `TS`) generates an output d (resp., e) for the input item b. As `PD` and `TS` are workers, the output d from `PD` carries signature $u0$, and e from `TS` carries signature $u1$. Therefore, we know d should be handled before e because $u0 < u1$.

Table 5.1 lists parallel patterns supported with the preservation of the sequential semantics in different tools. *Pipeline parallelism* involves breaking a task into a sequence of processing stages that are executed concurrently, with each stage taking the output of the previous stage as input and immediately passing its own output downstream. *Data parallelism* refers to the parallel execution of the same task on

disjoint parts of the data, such as the example shown in PT1 using parallel instances of the task `Map`. *MapReduce* is a widely used pattern consisting of three stages: map, shuffle, and reduce. This pattern typically partitions data streams by key, with partitioned sub-streams processed concurrently. *Task parallelism* involves concurrent execution of different tasks on multiple cores, and *non-linear task parallelism* refers to tasks that have non-linear dependencies, making them unable to be expressed as a simple pipeline. For example, PT3 contains non-linear task parallelism as `CMP` relies on outputs from both `PD` and `TS`.

StreamIt [55] restricts the language to a synchronous subset: it has copy and round-robin splitters but disallows value-based stream splitting. As a result, it is not able to support all forms of data parallelism and non-linear task parallelism. Additionally, StreamIt does not have native support for the MapReduce pattern, as the key-based partitioning can introduce dynamic item rates. Rx [127] does not have native support for non-linear task parallelism, and it is unable to preserve the data order when merging outputs from multiple workers. The paper [1] addresses the parallelization of tasks with dynamic item rates, however, it may not be able to support non-linear task parallelism as its order-restoration scheme using sequence numbers is not sufficient to distinguish the order of data items. Trill [106] supports parallelism through streaming generalizations of MapReduce operations, but it cannot handle pipeline parallelism with many stages or non-linear task parallelism. Furthermore, Trill's preservation of semantics for data parallelism and MapReduce is conditional as it assumes a total ordering of timestamps carried by data items. Timely Dataflow [82, 237] supports various forms of parallelism, and it allows the preservation of sequential semantics by emitting a synchronization timestamp for each data item. However, a limitation arises as the Timely Dataflow engine needs to wait for the advancement of timestamps for

each data item it processes. This mandatory waiting period can greatly restrict the scalability of the parallelization.

5.5 Preserve Semantics with Signatures

In this section, we will give a formal definition of the signature. We will then present the algorithms for different kinds of nodes to preserve the sequential semantics and analyze their complexity.

A stream is typically viewed as an unbounded sequence of data items; While there are certain streams that indeed terminate (e.g., when reading lines from a text file). We further generalize this notion of a stream (i.e., unbounded sequence) by allowing the occurrence of a distinguished symbol $\square$, called *end-of-stream marker*, that signals the termination of the stream. Languages like StreamQRE [115] and StreamQL [87] also provide such generalization as it naturally enables the temporal composition of streaming operators. We use a *signature* to represent the semantics-preserving processing order of each data item, which is a sequence of natural numbers assigned to each streaming data element. We define the type of signatures as Sig. We use ε to represent an empty signature and describe the concatenation of two signatures u and v as $u \cdot v$ (later we omit $\cdot$ for simplicity). The signature follows a lexicographic order (e.g., $\varepsilon < 0 < 01 < 1$), which we describe with a function $\mathtt{cmp} : \mathsf{Sig} \times \mathsf{Sig} \to \{-1, 0, 1\}$:

$$
\begin{aligned}
&\mathtt{cmp}(u, v) = 0, && \text{if } u = v = \varepsilon,\\
&\mathtt{cmp}(u, v) = -1, && \text{if } (u = \varepsilon \text{ and } v \neq \varepsilon) \text{ or } h(u) < h(v),\\
&\mathtt{cmp}(u, v) = 1, && \text{if } (v = \varepsilon \text{ and } u \neq \varepsilon) \text{ or } h(u) > h(v),\\
&\mathtt{cmp}(u, v) = \mathtt{cmp}(r(u), r(v)), && \text{if } h(u) = h(v),
\end{aligned}
$$

where $h : \mathsf{Sig} \to \texttt{Nat}$ returns the head of a non-empty signature, and $r : \mathsf{Sig} \to \mathsf{Sig}$ returns the remainder. For signatures u and v, we say $u < v$ if $\texttt{cmp}(u, v) = -1$, $u = v$ if $\texttt{cmp}(u, v) = 0$, and $u > v$ if $\texttt{cmp}(u, v) = 1$.

We describe streaming computations using dataflow graphs constructed by three kinds of nodes: splitter, worker, and merger. These nodes are connected by FIFO channels, and each kind performs a specific algorithm to manipulate the signature of each data item. We have described the rough idea of these algorithms in Section 5.4. We will now present the details.

Execution of Worker

Algorithm 1 presents the execution of the worker node and specifies the manipulation of the signature in the worker. The worker receives inputs from `input_ch`. For each input item, the worker uses `gen_vals` to generate the output values and `combine` to assign the signature to each output value.

Algorithm 1: The execution of the worker node.

```
Data: Input channel input_ch
while input stream is not terminated do
    in = input_ch.get() ;                    /* Get an input with signature */
    vals = gen_vals(in) ;                        /* Generate output values */
    forall val in vals do
        out = combine(val, in.sig);
        emit(out) ;                        /* Emit out to the output channel */
```

For example, as shown in Figure 5.9, we consider a graph that is a 3-stage pipeline of workers as shown below, where a, b, c are values of input data items, $\square$ is the end-of-stream marker, and subscripts u, v, w, x are signatures carried by input items. Worker w_1 in the pipeline echoes its input item to its output. Worker w_2 also echoes the input while halts early upon the arrival of c_w. Therefore, the end-of-stream marker

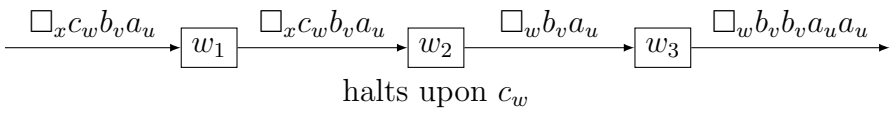


Figure 5.9 : Example of the execution of workers.

produced by w_2 carries the same signature as c_w. Worker w_3 echoes each input data item twice, where its output items, generated by the same input, have the same signature.

Execution of Splitter

Algorithm 2 shows the execution of the splitter node, and it presents how splitters assign signatures to each of its emitted outputs. The splitter will count the number of emitted outputs as n and extend the signature of each incoming input item by this number (i.e., `extend(in.sig, n)`). This extended signature will be assigned to the output item.

Algorithm 2: The execution of the splitter node.

```
Data: Input channel input_ch
n ← 0;
while input stream is not terminated do
    in = input_ch.get();
    vals = gen_vals(in);
    forall val in vals do
        new_sig = extend(in.sig, n);
        out = combine(val, new_sig);
        emit(out);
        n = n + 1;
```

For example, let us consider three kinds of common splitting strategies: round-robin, hashing, and data-copying, where the first two strategies are usually used in data parallelism, while the third one is typically used in task parallelism. Figure 5.10(a)

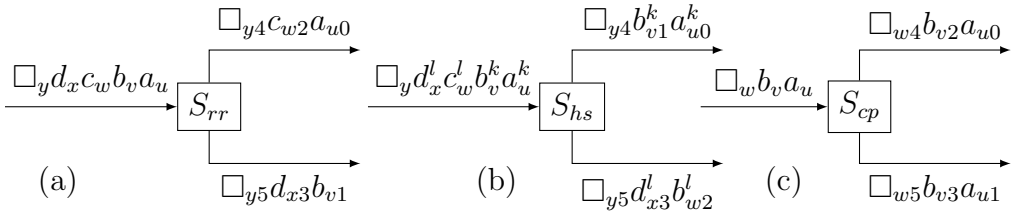


Figure 5.10 : Example of the execution of splitters.

shows a round-robin splitter s_{rr} that distributes input items in a round-robin way. Figure 5.10(b) shows a hashing splitter s_{hs} that splits input items based on their key identifiers, where the key identifier of item a^k_u (resp., c^l_w) is k (resp., l). Figure 5.10(c) presents a data-copy splitter s_{cp} that duplicates each input item and sends out copies through two output channels. If the splitter observes an end-of-stream marker, it will send it to all of its output channels to notify all of the connected nodes of the termination of the data stream.

Execution of Merger

A merger selects the item with the smallest signature from input items received from multiple input channels. Suppose this item has a signature u. The merger will drop the rightmost integer of u and assign it to each output generated by this item. Algorithm 3 presents how a merger manipulates the signatures of data items. For each input channel, we use a FIFO buffer to store received items, which is considered inactive if there is an end-of-stream marker at its head. A minimum heap, denoted as `hp`, is used to maintain the head of each buffer. The top of the heap tracks the data item with the smallest signature. When an input item arrives, the heap is updated by function `update`. Once an item is ready for processing (i.e., known to have the smallest signature), it is removed from the heap, and the function `trim` is used to

Algorithm 3: The execution of the merger node.

```
Data: Input channels input_chs[N]
rr ← 0 ;                                   /* round-robin index for reading inputs */
num_eos ← 0 ;                                        /* number of collected □s */
bufs ← Queue[N] ;                                /* buffers for all input channels */
/* MinHeap stores (k, v) pairs -- k is the head of an input channel, v is the
   index of this channel in input_chs.                                            */
hp ← MinHeap() ;                                      /* MinHeap is sorted by k.sig */
has_eos = bool[N] ;                        /* track if an input channel terminates */
num_active = N ;                                    /* number of active buffers */
while true do
    if has_eos[rr] then
        rr = (rr + 1) mod N;
        continue;
    in = input_chs[rr].get();
    if in is □ then
        has_eos[rr] = true;
        num_eos = num_eos + 1;
        if bufs[rr].size() == 0 then
            num_active = num_active − 1;
        update(hp, bufs, has_eos, num_active);
        if num_eos == N then
            new_sig = in.sig.trim();
            vals = gen_vals(in);
            forall val in vals do
                emit(combine(val, new_sig)) ;                          /* emit item */
            emit(Eos(new_sig)) ;                    /* emit eos with new signature */
            break;
    else
        buf = bufs[rr];
        buf.add(in) ;                                    /* now in is a data item */
        if in is at the head of buf then
            hp.add(in, rr);
            update(hp, bufs, has_eos, num_active);
Function update(hp, bufs, has_eos, num_active)
    while hp.size() != 0 and hp.size() == num_active do
        (top, index) = hp.poll();
        buf = bufs[index];
        buf.pop();
        new_sig = top.sig.trim();
        vals = gen_vals(data);
        forall val in vals do
            emit(combine(val, new_sig));
        if buf.size() > 0 then
            hp.add((buf.head(), index));
        else if has_eos[index] then
            num_active = num_active − 1;
```

obtain a new signature by removing the rightmost integer of the source signature. This new signature is then assigned to each output item generated by the input item.

Let us consider an example. Figure 5.11(a) presents the sequential execution of a computation `id` that echoes the input. Figure 5.11(b) shows a DAG that computes `id` with data parallelism, which includes a round-robin splitter s_{rr}, workers w_1 and w_2, and a merger m, where w_1 and w_2 concurrently computes `id`. The merger m collects output items from w_1 and w_2, reorders them by their signatures, and emits them once they are ready to be sent. Suppose m receives a_0 as the first input item, it can outputs a_0 immediately once seeing b_1 produced by w_2 that has a greater signature (i.e., a_0 is ready to be sent); Otherwise, a_0 will be buffered. Finally, if m receives $\square$, it will emit an $\square$ only if it collects $\square$s from all of its input channels. The output of m in Figure 5.11(b) is the same as the output as shown in Figure 5.11(a) – the sequential semantics is preserved.

As another example, Figure 5.11(c) shows the sequential computation, where we group input items by their keys, and we then duplicate each item into two copies and perform computations `id` and `last` respectively to each copy (`last` selects the last item in the stream). We use `f` // `g` to specify a computation: for each input item, we respectively perform computation `f` and `g` to the input and always emit the output of `f` before `g`. Therefore, in this example, the output produced by `id` should be handled before the output of `last`, e.g., $c^l_{\texttt{id}}$ (i.e., c^l produced by `id`) is handled before $c^l_{\texttt{last}}$. Finally, the outputs are aggregated by a binary function (the combinator is denoted by $\circ$). Figure 5.11(d) presents the parallel execution of this computation, which has the same output as shown in Figure 5.11(c).

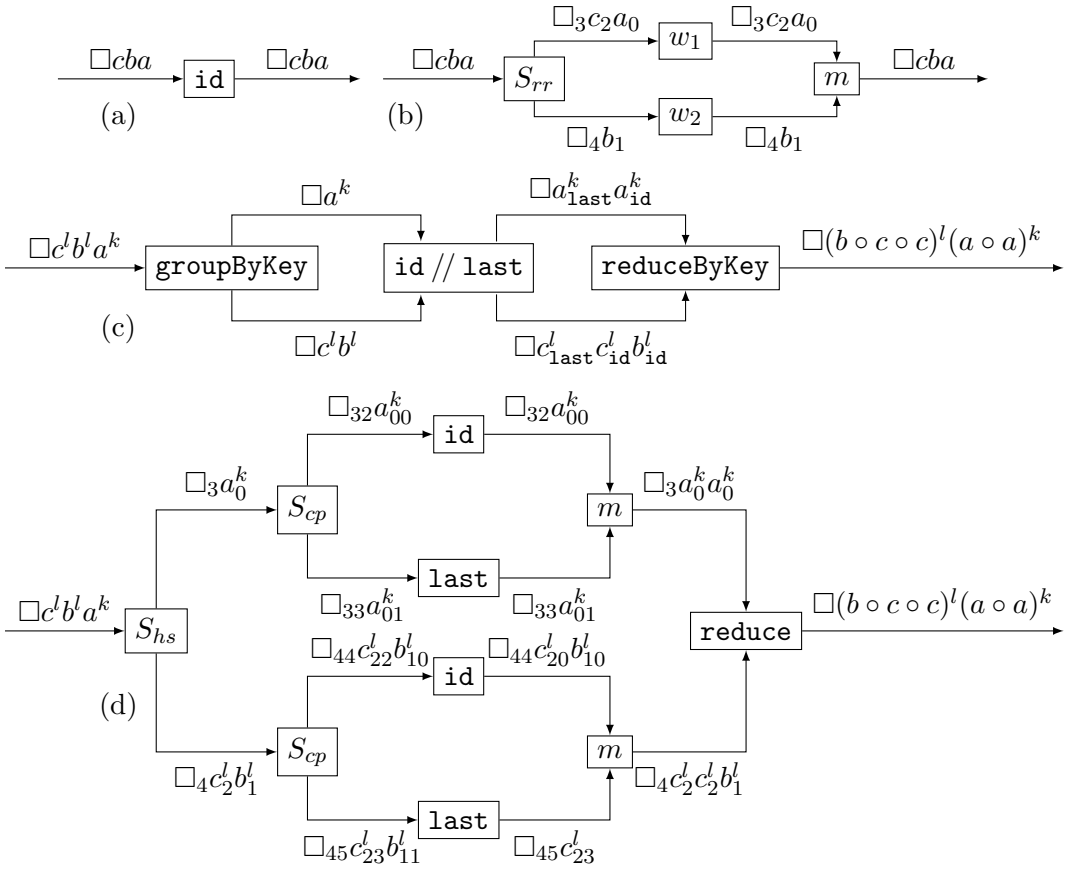


Figure 5.11 : Example of execution related to data parallelism.

5.5.1 Complexity of Signature Manipulation

The process of assigning a signature is relatively inexpensive in terms of complexity. For workers, the signature of the input item is directly copied as the signature of the corresponding output item(s). For splitters, a simple counter is used to count the number of emitted output items and the signature of the input is extended by this counter. In the case of mergers, the rightmost integer is dropped from the signature of the input, and the resulting value is assigned as the signature of the outputs. The time complexity of signature assignment is $O(1)$ per item and a constant amount of memory is used to store the signature of the input item and a counter if necessary.

However, merging signatures can be more computationally expensive as the merger needs to track the smallest signature received from all input channels. While the FIFO nature of the channels ensures the correct order of signatures inside each input channel, the merger must still compare the signatures of data items at the head of each input channel. The cost of the merging operation is non-deterministic as some items may arrive later to the input channel. In the worst-case scenario, if an input item never arrives at an input channel, the merger will buffer the entire data stream into memory and the output will never be emitted. To address this issue, we implement heartbeats in § 5.5.2 which ensures that an output can be emitted in a constant amount of time even in the absence of input items from certain input channels.

We define the *dimension* of signature to analyze the complexity of the merging algorithm. Each edge (data channel) of the DAG contains data items whose signatures are of the same length. We call such a length the dimension of the edge. For a splitter, if its input channel is of dimension d, all of its output edges will be of dimension $d + 1$ since the signature is extended by an integer. For a worker, the input edge and output edge share the same dimension as the signature of the input item is assigned to the

Table 5.2 : Time and space complexity of the signature manipulation per data item (k is the number of input edges, d is the dimension of the signature).

	splitter	worker	merger
time	$O(1)$	$O(1)$	$O(d \cdot \log k)$
space	$O(1)$	$O(1)$	$O(k)$

generated output items. For a merger, if its input edge is of dimension d, all its output edges will have a dimension of $d-1$. The time complexity of the lexicographic-order-based comparison between two signatures of dimension d is $O(d)$. Therefore, for a merger with k input edges, using brute force to compute the smallest signature would have a time complexity of $O(dk)$ per output item and a space complexity of $O(1)$. We use the minimum heap in Algorithm 3 to reduce the time complexity to $O(d \cdot \log k)$. The space complexity of the minimum heap is $O(k)$ as it stores the signature of the first item from each input channel.

Table 5.2 summarizes the complexity of signature manipulation per data item for different types of DAG nodes. It shows that assuming a constant number of input edges, typical for the limited number of computing cores on a machine, the memory usage for manipulating signatures is $O(1)$ per data item. The time complexity is close to $O(d_{max})$ per data item, where d_{max} is the maximum length of the signature, which is usually a small number.

5.5.2 Heartbeats

When merging inputs from multiple channels, we process an input item immediately upon determining that it has the smallest signature. Otherwise, we temporarily store it in a buffer. However, this approach can result in high latency and memory usage if an input item never arrives at a particular channel.

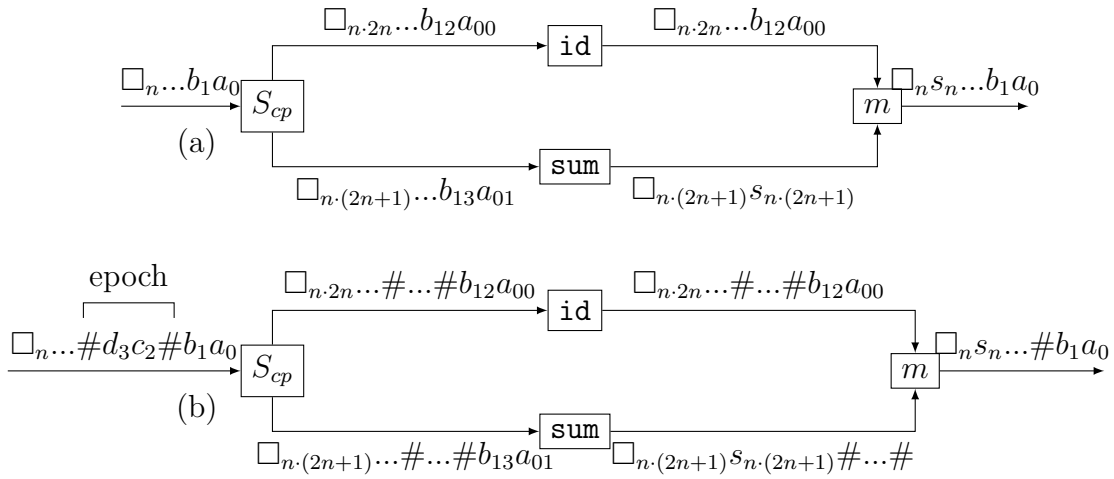


Figure 5.12 : Example of execution without and with heartbeats.

To address this problem, we incorporate a mechanism called *heartbeats* (denoted by #) into the data stream at regular intervals. Heartbeats act as special data items that represent the absence of items on a stream, similar mechanisms are often used in other systems under various names such as punctuation [240], watermarks [100], or pulses [1]. Each node propagates heartbeats by emitting them to each of its output channels once it has collected a heartbeat from all of its input channels. We call the duration between the insertion of heartbeats an *epoch*. Figure 5.12(a) and (b) respectively show the DAG for the computation `id` // `sum` without and with heartbeats, where the epoch is 2 and `sum` emits the sum of input items, denoted by s, upon the termination of the input stream. The use of heartbeats guarantees that the merger will receive data items on all input channels at least once per epoch. Therefore, a merger can emit buffered data once it has collected a heartbeat from each of its input channels, this ensures a low latency and memory usage.

5.6 Experimental Evaluation

We have implemented ParaStream as a Rust library to support semantics-preserving parallelism with our proposed algorithm. In this section, we will compare ParaStream against StreamQL [87], RxJava [205], Reactor [128], and Timely Dataflow [239], which are all lightweight high-performance libraries that can be used to process streaming data. StreamQL is designed for single-threaded processing, while RxJava, Reactor, and Timely Dataflow can be used in a multi-threaded setting. Notice that RxJava and Reactor lack support for non-linear task parallelism.

The experiments were executed in Ubuntu 20.04 on a desktop computer equipped with an Intel(R) Xeon(R) W-2295 CPU (18 cores) and 128 GB of RAM. We used OpenJDK 17 for Java programs and Rust 1.66-nightly for Rust programs. All data points represent the average of at least five runs, with error bars showing the standard deviation.

5.6.1 Evaluation of Sequential Implementation

We have evaluated the performance of the single-threaded implementation of ParaStream against StreamQL, RxJava, Reactor, and Timely Dataflow. We create an input stream of timestamped integers of the form $\{\texttt{ts}, \texttt{val}\}$ as "$\{1, 1\}$, $\{2, 2\}$, ..., $\{n, n\}$", where both the timestamp `ts` and the data value `val` are integers, and n is set to be 10 million. We run several basic streaming computations over such an input stream, and the queries are: `map` selects the value of each input item, `filter` removes items with odd integer values, and `sum` calculates the sum of the values. The qualifiers `twnd`, `swnd`, and `grp` refer to aggregation over tumbling windows, sliding windows, and key-based partitions respectively. The qualifier `gtw`(`gsw`) refers to tumbling (sliding) window aggregation over key-based partitions. For computations that involve

key-based partitioning, we set $\texttt{key}(x) = x.\texttt{val}\ \mathit{mod}\ 100$, and for windows, we always fix the window size to be 100 and the sliding interval to be 1 (if it is a sliding window).

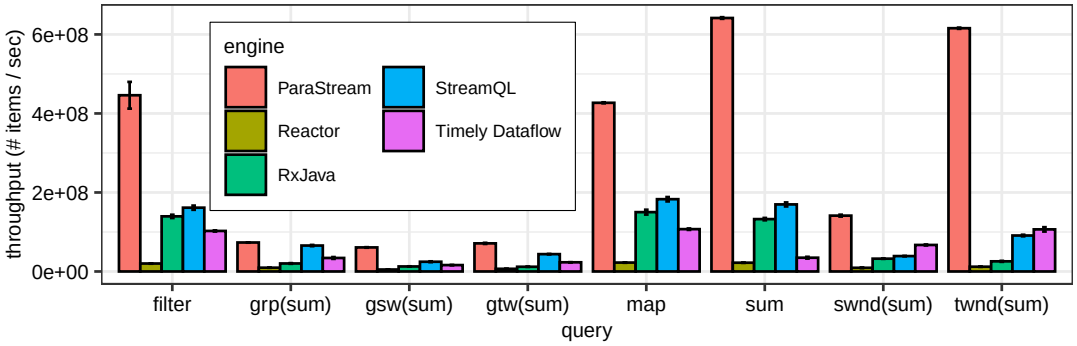


Figure 5.13 : Throughput (# items/sec, vertical axis) of ParaStream, Reactor, RxJava, StreamQL, and Timely Dataflow in single-threaded settings.

Figure 5.13 illustrates the throughput of ParaStream, Reactor, RxJava, StreamQL, and Timely Dataflow in single-threaded settings. ParaStream, adopting the technique introduced by StreamQL, avoids the overheads brought by the construction of nested stream objects, which makes it significantly faster than RxJava (3-25 times), Reactor (8-50 times), and Timely Dataflow (2-20 times). Moreover, ParaStream is 1.2-6 times faster than StreamQL.

5.6.2 Evaluation of Parallel Implementation

We have also evaluated the throughput performance of ParaStream in multi-threaded settings. We assigned one thread for each worker, as well as two threads for the stream splitter and merger. We used a function f, parameterized by n, which is defined as $f_n(x) = 3^n \cdot x$. This function allows us to evaluate the scalability of the throughput as the primitive operation f becomes increasingly compute-intensive (by increasing the parameter n).

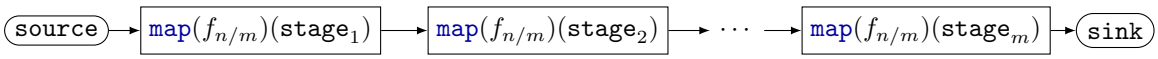


Figure 5.14 : The dataflow graph for evaluating the throughput of pipeline parallelism.

Evaluation of the Throughput of Pipeline Parallelism

We consider using a pipeline of m stages to perform the computation map(f_n), where map(f_n), for each input value x, computes the output value as $f_n(x)$. The computational workload is partitioned uniformly across the pipeline stages – each stage performs the computation map($f_{n/m}$). In the experiment, as n may not be exactly divisible by m, each of the first $m-1$ pipeline stages performs map(f_a), where $a = \texttt{floor}(n/m)$, and the last stage computes map(f_b), where $b = n - (m-1) \cdot a$. Figure 5.14 shows the dataflow graph used for evaluating the throughput of pipeline parallelism. In the ParaStream implementation, we created a worker thread to execute the computation for each pipeline stage. We evaluated the performance of ParaStream using the naïve algorithm without introducing signatures (i.e., preserve semantics by the FIFO property of data channels) and the general semantics-preserving algorithm (assign and copy signatures).

Figure 5.15(a) and (b) show the scalability of throughput in ParaStream without and with the use of signature respectively. The baseline in these figures represents the performance of the sequential implementation, with a speedup of 1. We also measured the overhead caused by the propagation of signatures as the throughput reduction per worker, as shown in Figure 5.15(c). Suppose the throughput of the implementation without (resp., with) the use of signature is t_1 (resp., t_2). The throughput reduction α is computed as: $\alpha = (t_1 - t_2)/t_1$. The results indicate that the overhead is less than 2 % for each worker. Additionally, Figure 5.15(d), (e), and (f) present the throughput

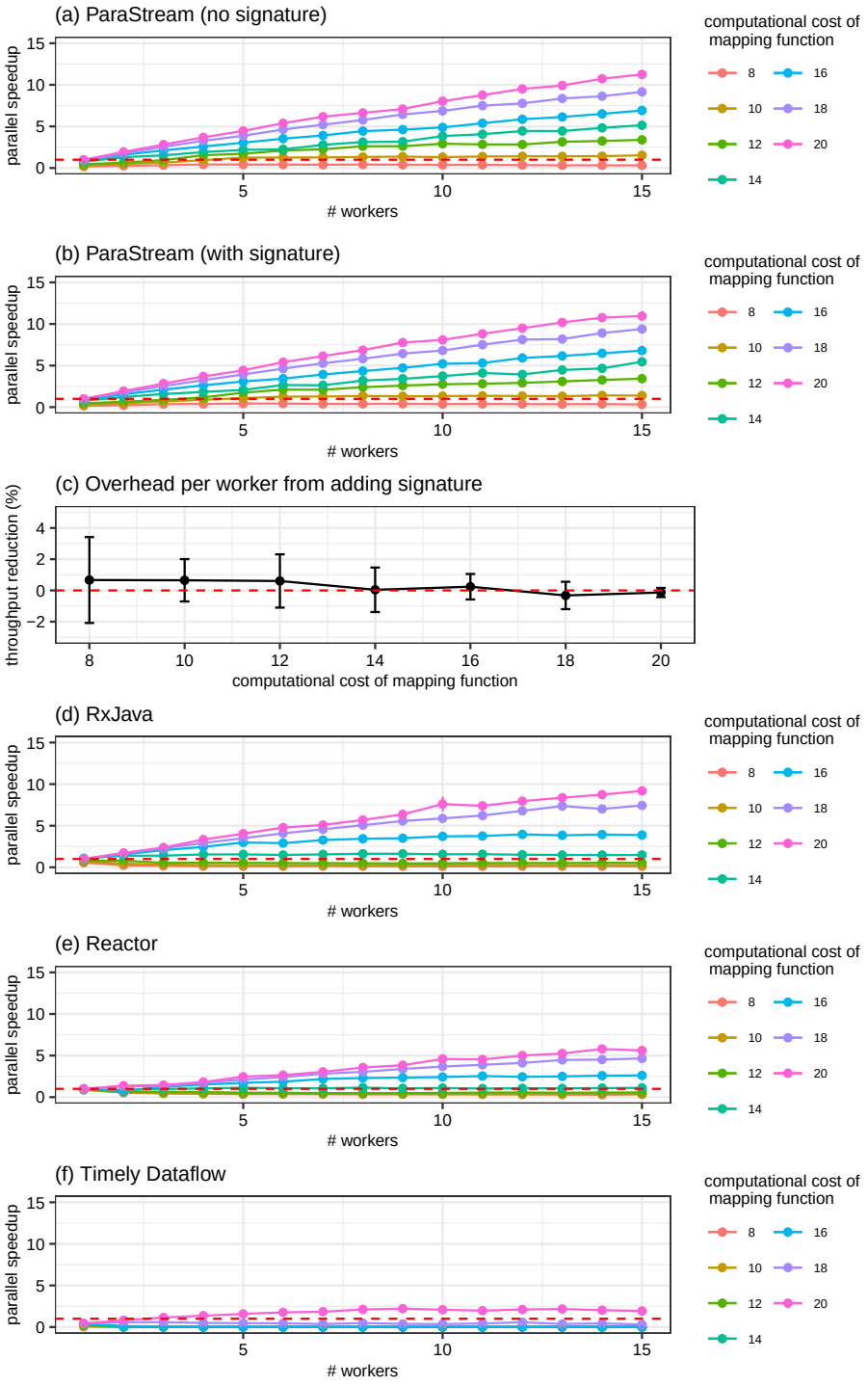


Figure 5.15 : The throughput scalability of the pipeline parallelism implemented in ParaStream, RxJava, Reactor, and Timely Dataflow.

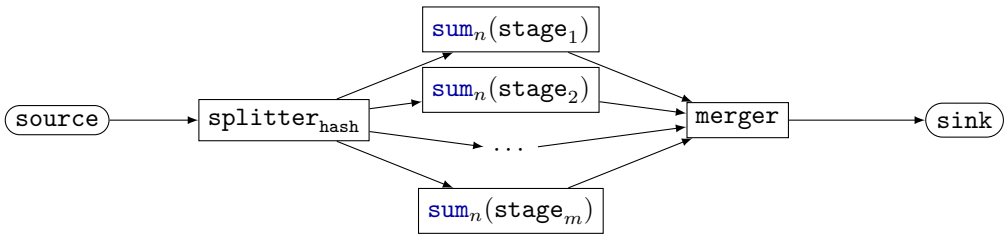


Figure 5.16 : The dataflow graph for evaluating the throughput of data parallelism.

scalability of implementations in RxJava, Reactor, and Timely Dataflow. We observe that ParaStream provides better scalability in terms of throughput.

Evaluation of the Throughput of Data Parallelism

We have also evaluated the scalability of throughput in terms of data parallelism – the input stream is partitioned into multiple sub-streams, and each worker handles a sub-stream of the input. We used query groupBy(sum$_n$) to describe the computation of key-based aggregation, where sum$_n$ is a binary function given by sum$_n(agg, x) = agg + f_n(x)$ that updates the aggregate. Recall that the parameter n controls the computational cost of f_n and hence sum$_n$. To evaluate the parallel implementation of this query, we used a splitter thread to partition the input stream equally with the key-extraction function $key(x) = x \bmod 256$ for each input integer x. We sent send items with the same key to the same worker thread and used the worker thread to perform sum$_n$ for each key. The results are then collected by a merger thread. Figure 5.16 shows the data flow graph for implementing such a pattern. We have evaluated the performance of three implementations: (I) one that allows the worker threads to output items as they become available (i.e., no order), (II) one that preserves the sequential semantics by directly reordering aggregation results based on values of their keys (i.e., efficient algorithm), and (III) one that using the general semantics-preserving algorithm (i.e.,

general algorithm). Furthermore, we compared the performance of ParaStream against RxJava and Reactor, which do not preserve the semantics.

Figure 5.17(a), (b), and (c) show the scalability of implementation (I), (II), and (III) respectively, and Figure 5.17(d) presents the overheads (ratio of throughput slowdown per worker compared to implementation (I)) brought by the preservation of semantics. As seen in Figure 5.17(d), our reordering algorithm adds negligible overhead ($< 2\,\%$) to the throughput of each worker. Figure 5.17(e), (f), and (g) show the scalability of throughput for RxJava, Reactor, and Timely Dataflow, which do not preserve the sequential semantics. By comparing Figure 5.17(c), (e), (f), and (g), we can observe that ParaStream provides better throughput scalability in terms of the implementation of data parallelism compared to RxJava, Reactor, and Timely Dataflow, even though it incurs additional overheads to preserve the sequential semantics.

Evaluation of the Throughput of Task Parallelism

We considered a computation that collects the results of 120 tasks for each input, where each task computes `map`(f_n) over the input value. In this computation, given a stream that contains l input items, an output stream with $120 \cdot l$ items will be produced. To parallelize this computation, we used a splitter to send a copy of each input item to multiple workers, with the number of tasks uniformly partitioned across workers. Figure 5.18 shows the corresponding dataflow graph. Suppose the number of workers is m, each worker will handle $120/m$ tasks. Finally, we used a merger to collect outputs from these workers. We have evaluated three implementations: (I) one that allows the worker threads to output items as they become available (i.e., the no-order implementation), (II) one that preserves the sequential semantics by buffering outputs until each worker generates $120/m$ outputs (i.e., the efficient implementation),

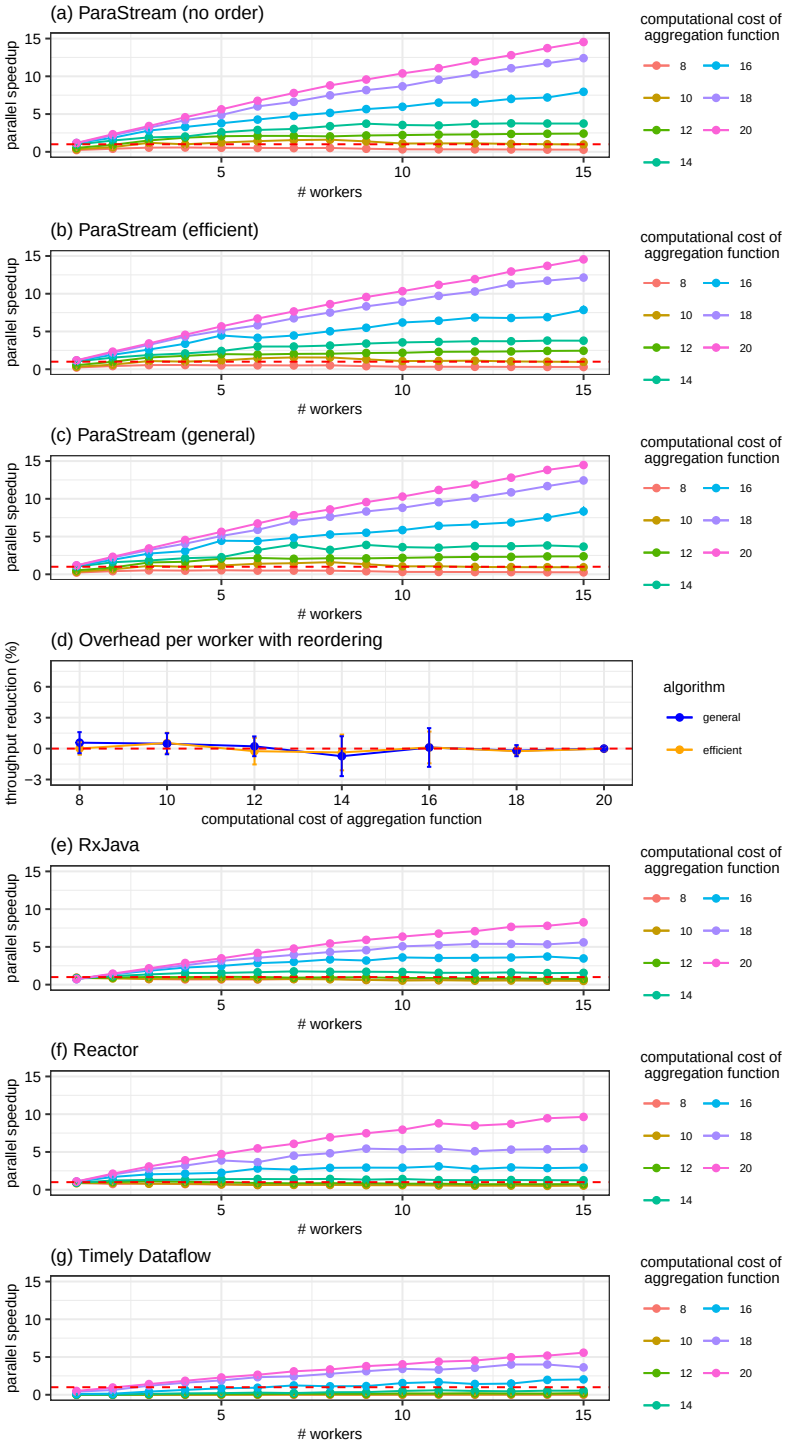


Figure 5.17 : The throughput scalability of the data parallelism implemented in ParaStream, RxJava, Reactor, and Timely Dataflow.

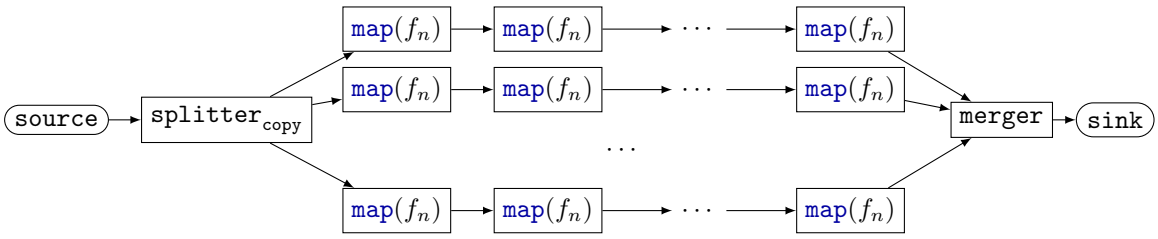


Figure 5.18 : The dataflow graph for evaluating the throughput of task parallelism.

and (III) one that uses the general semantics-preserving algorithm as described in Section 5.5 (i.e., the general implementation).

Figure 5.19(a), (b), and (c) show the throughput scalability of implementation (I), (II), and (III) respectively, and Figure 5.19(d) presents the overheads caused by the preservation of semantics – we observe the efficient algorithm introduces negligible throughput slowdown, while the general algorithm brings small overheads ($< 5\%$ slowdown per worker). Figure 5.19(e) shows the scalability of the implementation in Timely Dataflow, where the implementation does not preserve the sequential semantics. In the comparison between Figure 5.19(c) and Figure 5.19(e), we observe our ParaStream implementation provides better throughput scalability against Timely Dataflow even paying additional overheads to preserve the semantics. Notice we do not evaluate the throughput of RxJava and Reactor for implementing task parallelism because they do not natively support such a pattern.

Impact of Heartbeat

In reference to Section 5.5.2, an epoch is defined as the number of data items on each channel between two consecutive heartbeats. We have evaluated the impact of increasing the epoch on the speedup of throughput for the parallel computation of map(f_n). We utilized a splitter to divide the input stream into several sub-streams,

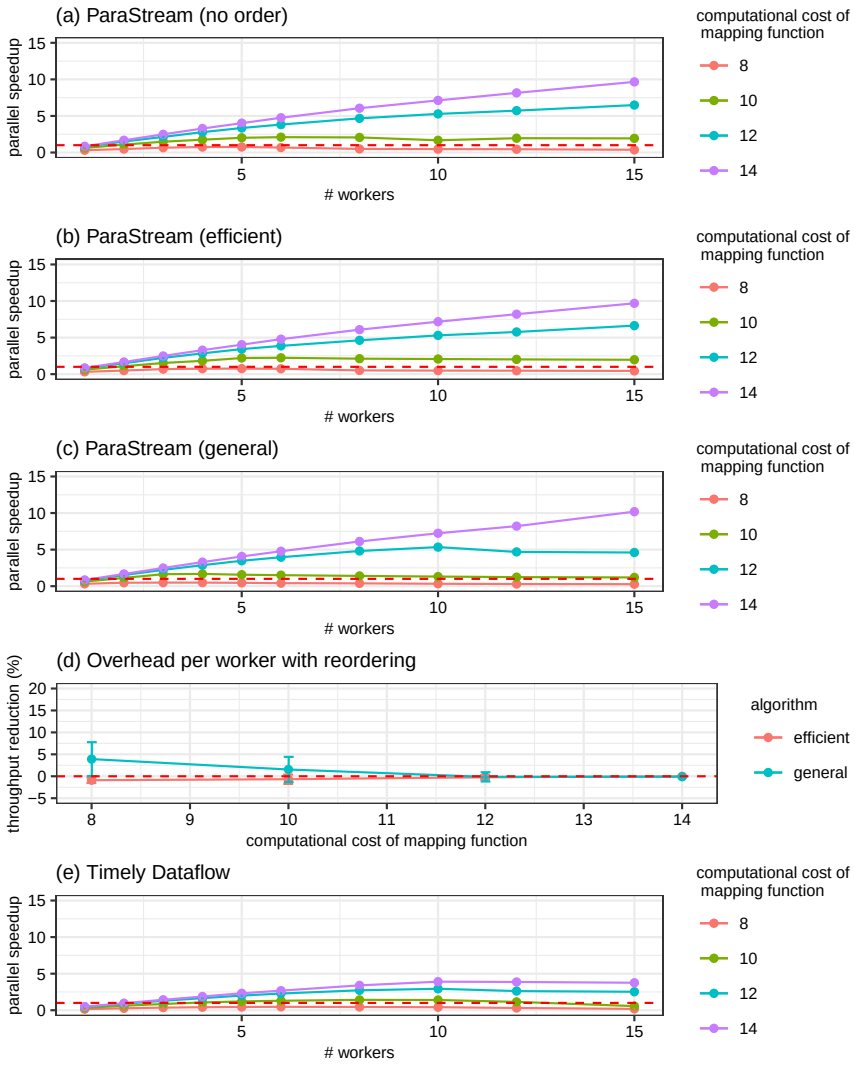


Figure 5.19 : The throughput scalability of the task parallelism implemented in ParaStream and Timely Dataflow.

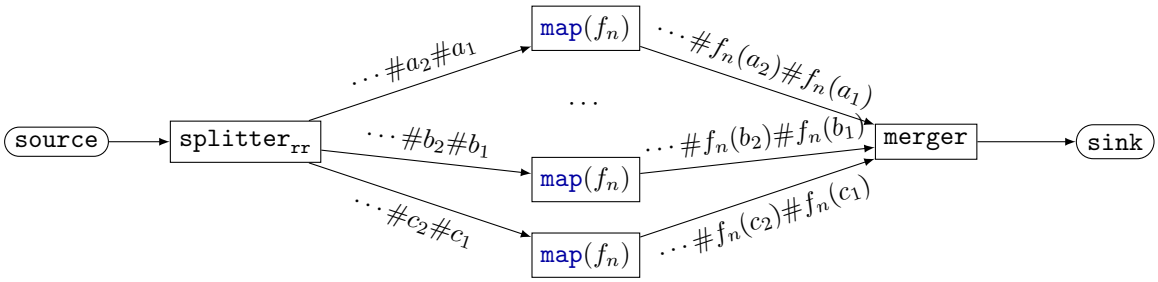


Figure 5.20 : Example of the dataflow graph for evaluating the impact of heartbeats (epoch = 1).

each of which was assigned to a worker for computation. The results were then collected by a merger in a way that preserved the semantics. For an epoch of e, the splitter would send ew items to each worker before sending a heartbeat, where w is the number of workers. Figure 5.20 shows an example of the dataflow graph for examining the impact of the insertion of heartbeats. We examined epochs of $e = 1, 4, 16, 64, 256$.

The scalability of the throughput for parallelizing `map`(f_n) with different epochs can be seen in Figure 5.21(a), (b), (c), (d), and (e), and the overhead caused by adding heartbeats is illustrated in Figure 5.21(f). As the value of the epoch increases, the overhead decreases due to the reduction in the frequency of heartbeat synchronization.

Real-world Benchmarks

We have also evaluated the throughput scalability of ParaStream on the following real-world workloads:

regex. The regex matching benchmark involves counting the occurrences of patterns described by regular expressions in an input text. We iterated a 2 MB English book as the input source, which was fed over a 10-sec sliding window with a 1-sec interval.

urlcnt. The URL counting benchmark [241] requires the parsing of network packets and the counting of unique URL identifiers inside the packets. We used the Yandex dataset [242] with 70 M unique URLs and a 30-sec tumbling window.

pattern. The stock pattern detection benchmark [221, 58, 57] involves the detection of five consecutive quotes whose prices fluctuate in a V-pattern (down, down, up, up) for each stock in the market. We used the dataset from [220] and considered a 1-min tumbling window.

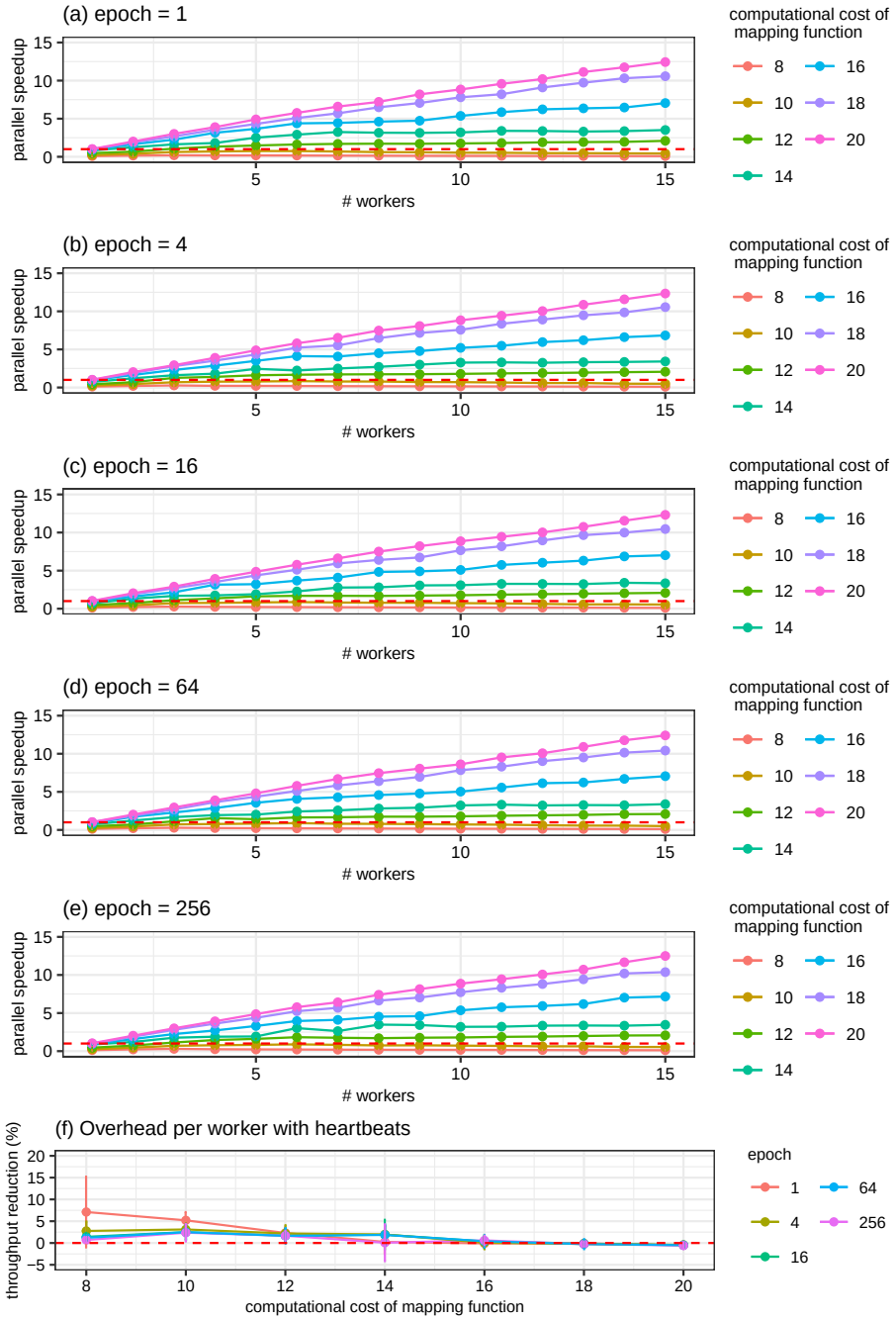


Figure 5.21 : The throughput scalability for parallelizing `map`(f_n) with different epochs in ParaStream.

netmon. The network latency monitoring benchmark [241, 191] requires the partitioning of network latency records by IP pairs (i.e., source-destination) and the computation of the average latency per partition. We used the dataset introduced in [243] and a 10-sec tumbling window.

loadpred. The load prediction benchmark involves the computation of the median value of the load measurement (in Watts) in a 30-min sliding window for each household, which is a variant of the benchmark used for the "Smart Homes" competition of the DEBS 2014 conference [244].

bbo. The best bid and offer matching [87] benchmark searches for the best bid and offer from high-frequency stock transactions for each stock. We used data from the NYSE TAQ dataset [80], which collects real-time trades and quotes reported on the U.S. Consolidated Tape and fed the data over a 100-ms tumbling window.

rsi. This benchmark involves the computation of the relative strength index (RSI) for each stock. RSI [245] is a technical indicator used in financial markets to measure the price momentum of a stock or other security. The RSI calculation [246] involves comparing the average fluctuation of trading prices and volumes over a certain number of periods. We also used NYSE TAQ dataset [80] to compute RSI.

pagerank. The PageRank benchmark [82] involves the iterative ranking of pages in a web graph using a feedback loop. We multiplied the rank vector with the web graph adjacency matrix in each iteration, where each worker multiplies the previous rank vector with some rows of the adjacency matrix. and their results will be later assembled by a merger.

kmeans. The K-Means benchmark involves iterative computation for unsupervised clustering using the K-Means algorithm [247]. In each iteration, a splitter will assign samples and initial centroids to workers. Each worker will update the local centroids based on the samples it receives. At the end of each iteration, workers send their local centroids to a merger, which will summarize the results and sends them back to the splitter. We used a synthesized dataset containing 10,000 samples in 10 clusters, where each sample consists of 100 features.

har. The Human activity recognition benchmark involves the streaming classification of human motions (standing, sitting, laying down, walking, etc.) by convolutional neural networks (CNN) [248] inference. We used data from a dataset [249] that contains signals collected from smartphones. To perform the CNN inference, we used sliding windows of size 2.65 seconds as suggested in [249].

vibration. The vibration monitoring benchmark involves monitoring the vibration signals of ball bearings to predict their failure rates. These sensors generate data streams at a very high frequency, often as much as 40 KHz [250]. To accurately predict failures, statistical analysis tools such as kurtosis [251] and crest factor [252] can be computed on the signal measurements, where kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution, and crest factor is a parameter of a waveform, such as alternating current or sound, showing how extreme the peaks are in a waveform. We used a real-world dataset [253] collected from bearings with different health conditions. We performed analysis over a 100-msec tumbling window.

fraud. The fraud detection benchmark involves the identification of fraudulent activities from the financial transaction data of their customers. The strategy

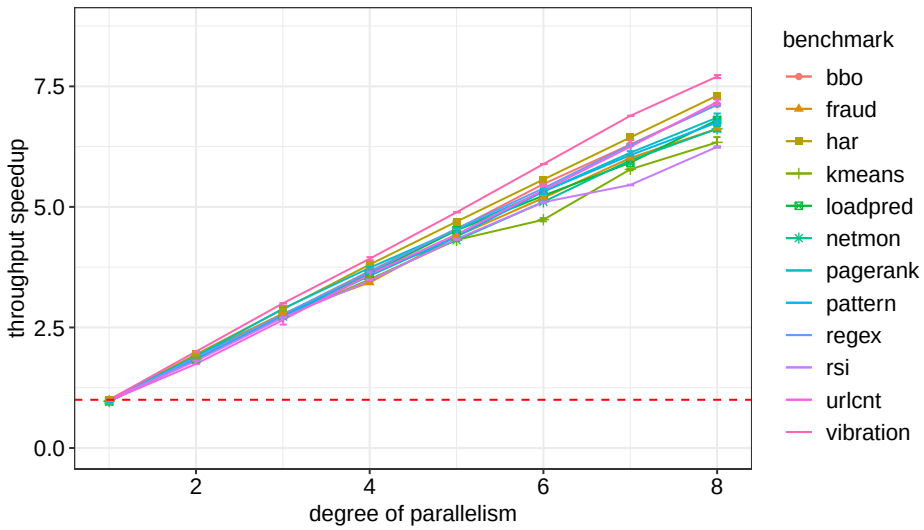


Figure 5.22 : Throughput speedup for real-world benchmarks with different degrees of parallelism.

we used is to look for small payment transactions with quantity p_s, followed by large payment transactions with quantity p_l, occurring within a short time interval t [254]. We computed the moving average (μ) and standard deviation (σ) of the transaction amounts for each customer over a 10-day sliding window. The threshold for large (resp., small) quantity is set as $\mu + 3\sigma$ (resp., μ - 3σ).

The scalability of throughput for the ParaStream implementations that handle the aforementioned real-world workloads while preserving semantics can be seen in Figure 5.22. The horizontal axis represents the degree of parallelism, which is the number of threads used to perform the computation on workers. It can be observed that there is a near-linear increase in throughput compared to the sequential implementation for different real-world workloads.

5.7 Chapter Summary

In this chapter, we present a programming system for safely parallelizing the processing of streaming data on a multicore CPU. Our system is based on the use of signature, allowing the preservation of sequential semantics in the implementation of complex patterns of parallelism. We have developed a Rust library called ParaStream to support semantics-preserving parallelism in the processing of data streams. Our experimental results show that ParaStream, in terms of single-threaded throughput, consistently outperforms state-of-the-art tools. ParaStream is 1.2 to 6 times faster than StreamQL, 2 to 20 times faster than Timely Dataflow, 3 to 25 times faster than RxJava, and 8 to 50 times faster than Reactor. Additionally, ParaStream offers superior throughput scalability compared to RxJava, Reactor, and Timely Dataflow. Furthermore, ParaStream provides substantial performance gains with increasing degree of parallelism in real-world benchmarks.

Chapter 6

Conclusion

6.1 Summary

This thesis discusses the language support for real-time data processing. The technical sections of this thesis can be divided into three parts.

In Chapter 3, we introduced StreamQL, a language for specifying complex streaming computations as compositions of stream transformations. StreamQL merges relational, dataflow, and temporal constructs, offering a high-level approach for programming streaming analyses. StreamQL consistently outperforms popular streaming engines like RxJava, Reactor, and Siddhi across various real-world benchmarks, proving its efficacy.

In Chapter 4, we focused on hardware acceleration for regular pattern matching with regexes that include bounded repetitions. We formulated a design that integrates counter and bit vector modules into an in-memory NFA-based hardware architecture, inspired by the theoretical model of nondeterministic counter automata. Our regex-to-hardware compiler analyzes counter-(un)ambiguity over regexes and then generates an automaton representation that can be deployed on the hardware. This approach surpasses unfolding solutions significantly. In experiments with real-world benchmarks, we have observed significant energy and area reductions provided by our design compared to a state-of-the-art processor for in-memory regex matching.

In Chapter 5, we presented a novel programming system to parallelize the processing

of streaming data with the preservation of sequential semantics. Our algorithms utilize signatures to preserve the sequential semantics of the computations. We have implemented the programming system as a Rust library, ParaStream, which offers superior throughput and scalability to the existing tools.

6.2 Future Directions

Moving forward, there are several areas that I believe require further exploration.

Firstly, refining the existing scope of case studies is crucial for advancing the development of Domain-Specific Languages (DSLs) for stream processing. A recent study [255] has exposed flaws in current time series processing benchmarks, where those flawed benchmarks are often used to motivate the development of DSLs for stream processing. By examining streaming computations in real-world applications more closely, we can enhance the utility and effectiveness of stream processing DSLs.

Secondly, there is a lack of implementations of DSLs in low-level programming languages such as C/C++, Rust, etc. Such implementations are particularly beneficial for devices with limited computing resources, which are common in IoT systems. While existing high-level tools like Trill [106, 111], StreamQL [87] and LifeStream [132] offer substantial functionality, their requirement for memory-intensive installations of virtual machines like CLR in .NET or JVM in Java makes them less suitable for devices with memory constraints. There are some solutions implemented in C++ like RaftLib [256] and StreamBox [191]. However, they are not focused on such constrained devices. In this thesis, we have initiated the development of a lightweight Rust library. Future work will involve further investigation into its application on embedded devices, including FPGA for IoT use cases.

Last but not least, we can delve deeper into hardware-software co-design for efficient

regex matching. This thesis discusses the hardware-software codesign for efficient regex matching, which is inspired by the theoretical model of NCAs. In another work, we have developed a software regex matcher called BVA-Scan [89], inspired by the model of nondeterministic bit vector automata (NBVAs), where NBVAs are expressively equivalent to the NCAs if the counter is bounded. Following the development of BVA-Scan, we see the potential for a corresponding hardware architecture for regex matching based on the model of NBVAs. This could potentially bring further reductions in energy and memory usage.

Bibliography

[1] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Safe data parallelism for general streaming," *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 504–517, 2015.

[2] J. Lu, D. Birru, and K. Whitehouse, "Using simple light sensors to achieve smart daylight harvesting," in *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, BuildSys '10, (New York, NY, USA), p. 73–78, Association for Computing Machinery, 2010.

[3] M. Jia, A. Komeily, Y. Wang, and R. S. Srinivasan, "Adopting Internet of Things for the development of smart buildings: A review of enabling technologies and applications," *Automation in Construction*, vol. 101, pp. 111–126, 2019.

[4] Y. Ichimaru and G. B. Moody, "Development of the polysomnographic database on CD-ROM," *Psychiatry and Clinical Neurosciences*, vol. 53, no. 2, pp. 175–177, 1999.

[5] J. Kim, A. S. Campbell, B. E.-F. de Ávila, and J. Wang, "Wearable biosensors for healthcare monitoring," *Nature biotechnology*, vol. 37, no. 4, pp. 389–406, 2019.

[6] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM Infosphere Streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM*

SIGMOD International Conference on Management of Data, SIGMOD '10, (New York, NY, USA), pp. 1093–1104, ACM, 2010.

[7] D. Oladimeji, K. Gupta, N. A. Kose, K. Gundogan, L. Ge, and F. Liang, "Smart transportation: An overview of technologies and applications," *Sensors*, vol. 23, no. 8, 2023.

[8] P. D. Diamantoulakis, V. M. Kapinas, and G. K. Karagiannidis, "Big data analytics for dynamic energy management in smart grids," *Big Data Research*, vol. 2, no. 3, pp. 94–101, 2015. Big Data, Analytics, and High-Performance Computing.

[9] M. N. Nafees, N. Saxena, A. Cardenas, S. Grijalva, and P. Burnap, "Smart grid cyber-physical situational awareness of complex operational technology attacks: A review," *ACM Comput. Surv.*, vol. 55, feb 2023.

[10] Y. Park, R. King, S. Nathan, W. Most, and H. Andrade, "Evaluation of a high-volume, low-latency market data processing system implemented with IBM middleware," *Software: Practice and Experience*, vol. 42, no. 1, pp. 37–56, 2012.

[11] D. Shah, H. Isah, and F. Zulkernine, "Stock market analysis: A review and taxonomy of prediction techniques," *International Journal of Financial Studies*, vol. 7, no. 2, 2019.

[12] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: A stream database for network applications," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 647–651, ACM, 2003.

[13] A. D'Alconzo, I. Drago, A. Morichetta, M. Mellia, and P. Casas, "A survey on big data for network traffic monitoring and analysis," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 800–813, 2019.

[14] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[15] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma, "Query processing, approximation, and resource management in a data stream management system," in *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, www.cidrdb.org, 2003.

[16] D. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, C. Erwin, E. Galvez, M. Hatoun, A. Maskey, A. Rasin, A. Singer, M. Stonebraker, N. Tatbul, Y. Xing, R. Yan, and S. Zdonik, "Aurora: A data stream management system," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 666–666, ACM, 2003.

[17] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous dataflow processing for an uncertain world," in *Proceedings of the First Biennial Conference on Innovative Data Systems Research (CIDR '03)*, 2003.

[18] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The design of the Borealis stream processing engine," in *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR '05)*, pp. 277–289, 2005.

[19] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, 2006.

[20] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy, "Storm @ Twitter," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), pp. 147–156, ACM, 2014.

[21] S. Kulkarni, N. Bhagat, M. Fu, V. Kedigehalli, C. Kellogg, S. Mittal, J. M. Patel, K. Ramasamy, and S. Taneja, "Twitter Heron: Stream processing at scale," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), pp. 239–250, ACM, 2015.

[22] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 423–438, ACM, 2013.

[23] E. Wu, Y. Diao, and S. Rizvi, "High-performance complex event processing over

streams," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, (New York, NY, USA), p. 407–418, Association for Computing Machinery, 2006.

[24] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: A high-performance event processing engine," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (New York, NY, USA), p. 1100–1102, Association for Computing Machinery, 2007.

[25] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby, "Pattern matching in sequences of rows," tech. rep., IBM, 2007. ANSI Standard Proposal.

[26] M. Hirzel, "Partition and compose: Parallel complex event processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, (New York, NY, USA), pp. 191–200, ACM, 2012.

[27] M. Bucchi, A. Grez, A. Quintana, C. Riveros, and S. Vansummeren, "CORE: A complex event recognition engine," *Proc. VLDB Endow.*, vol. 15, p. 1951–1964, may 2022.

[28] "FlinkCEP - complex event processing for Flink." Available at `https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/libs/cep/`, 2023. [Online; Accessed 1 July, 2023].

[29] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[30] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice, "LUSTRE: A declarative language for real-time programming," in *Proceedings of the 14th ACM SIGACT-*

SIGPLAN Symposium on Principles of Programming Languages, POPL '87, (New York, NY, USA), pp. 178–188, ACM, 1987.

[31] G. Berry and G. Gonthier, "The Esterel synchronous programming language: Design, semantics, implementation," *Science of Computer Programming*, vol. 19, no. 2, pp. 87–152, 1992.

[32] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, and R. Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, 2003.

[33] K. Havelund and G. Roşu, "Efficient monitoring of safety properties," *International Journal on Software Tools for Technology Transfer*, vol. 6, no. 2, pp. 158–173, 2004.

[34] B. D'Angelo, S. Sankaranarayanan, C. Sanchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna, "LOLA: Runtime monitoring of synchronous systems," in *Proceedings of the 12th International Symposium on Temporal Representation and Reasoning (TIME'05)*, (New York, NY, USA), pp. 166–174, IEEE, June 2005.

[35] P. Thati and G. Roşu, "Monitoring algorithms for metric temporal logic specifications," *Electronic Notes in Theoretical Computer Science*, vol. 113, pp. 145–162, 2005. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).

[36] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07).

[37] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, vol. 51, pp. 5–30, Aug 2017.

[38] A. Chattopadhyay and K. Mamouras, "A verified online monitor for metric temporal logic with quantitative semantics," in *RV 2020* (J. Deshmukh and D. Ničković, eds.), vol. 12399 of *Lecture Notes in Computer Science*, (Cham), pp. 383–403, Springer, 2020.

[39] E. Meijer, "Your mouse is a database," *Commun. ACM*, vol. 55, p. 66–73, May 2012.

[40] C. Elliott and P. Hudak, "Functional reactive animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, (New York, NY, USA), pp. 263–273, ACM, 1997.

[41] A. Courtney, "Frappé: Functional reactive programming in Java," in *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages (PADL '01)* (I. V. Ramakrishnan, ed.), (Berlin, Heidelberg), pp. 29–44, Springer Berlin Heidelberg, 2001.

[42] I. Maier and M. Odersky, "Deprecating the observer pattern with Scala.React," tech. rep., EPFL, 2012.

[43] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, (New York, NY, USA), pp. 93–102, ACM, 2006.

[44] H. Barringer, A. Goldberg, K. Havelund, and K. Sen, "Rule-based runtime verification," in *VMCAI 2004*, vol. 2937 of *Lecture Notes in Computer Science*, (Heidelberg), pp. 44–57, Springer, 2004.

[45] RE2, "RE2: Google's regular expression library." Available at `https://github.com/google/re2`, 2023. [Online; Accessed 30 June, 2023].

[46] P. Hazel and Z. Herczeg, "PCRE2: Perl compatible regular expressions v2." Available at `https://www.pcre.org/`, 2023. [Online; Accessed 30 June, 2023].

[47] X. Wang, Y. Hong, H. Chang, K. Park, G. Langdale, J. Hu, and H. Zhu, "Hyperscan: A fast multi-pattern regex matcher for modern CPUs," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI '19)*, (Boston, MA), pp. 631–648, USENIX Association, 2019.

[48] H. Liu, S. Pai, and A. Jog, "Why GPUs are slow at executing NFAs and how to make them faster," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, (New York, NY, USA), pp. 251–265, ACM, 2020.

[49] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *The 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, (New York, NY, USA), pp. 227–238, IEEE, 2001.

[50] Z. K. Baker and V. K. Prasanna, "Time and area efficient pattern matching on FPGAs," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, (New York, NY, USA), p. 223–232, ACM, 2004.

[51] N. Tuck, T. Sherwood, B. Calder, and G. Varghese, "Deterministic memory-efficient string matching algorithms for intrusion detection," in *IEEE INFOCOM 2004*, vol. 4, (New York, NY, USA), pp. 2628–2639 vol.4, IEEE, 2004.

[52] B. C. Brodie, D. E. Taylor, and R. K. Cytron, "A scalable architecture for high-throughput regular-expression pattern matching," *ACM SIGARCH computer architecture news*, vol. 34, no. 2, pp. 191–202, 2006.

[53] Y. Huang, Z. Chen, D. Li, and K. Yang, "CAMA: Energy and memory efficient automata processing in content-addressable memories," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, (New York, NY, USA), pp. 25–37, IEEE, 2022.

[54] P. Dlugosch, D. Brown, P. Glendenning, M. Leventhal, and H. Noyes, "An efficient and scalable semiconductor architecture for parallel automata processing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, pp. 3088–3098, 2014.

[55] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the 11th International Conference on Compiler Construction (CC '02)* (R. N. Horspool, ed.), vol. 2304 of *Lecture Notes in Computer Science*, (Berlin, Heidelberg), pp. 179–196, Springer, 2002.

[56] E. Bouillet, R. Kothari, V. Kumar, L. Mignet, S. Nathan, A. Ranganathan, D. S. Turaga, O. Udrea, and O. Verscheure, "Processing 6 billion CDRs/day: From research to production (experience report)," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, (New York, NY, USA), pp. 264–267, ACM, 2012.

[57] B. Chandramouli, J. Goldstein, and D. Maier, "High-performance dynamic pattern matching over disordered streams," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 220–231, 2010.

[58] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 147–160, ACM, 2008.

[59] G. Cugola and A. Margara, "TESLA: A formally defined event specification language," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, (New York, NY, USA), p. 50–61, Association for Computing Machinery, 2010.

[60] H. Zhang, Y. Diao, and N. Immerman, "On complexity and optimization of expensive queries in complex event processing," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), p. 217–228, Association for Computing Machinery, 2014.

[61] B. Zhao, H. van der Aa, T. T. Nguyen, Q. V. H. Nguyen, and M. Weidlich, "EIRES: Efficient integration of remote data in event stream processing," in *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, (New York, NY, USA), p. 2128–2141, Association for Computing Machinery, 2021.

[62] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Communications of the ACM*, vol. 18, no. 6, pp. 333–340, 1975.

[63] "GNU grep - Global Regular Expression Print." Available at `https://www.gnu.org/software/grep/`, 2022. [Online; Accessed 30 June, 2023].

[64] "GNU Awk." Available at `https://www.gnu.org/software/gawk/`, 2023. [Online; Accessed 30 June, 2023].

[65] I. Roy and S. Aluru, "Discovering motifs in biological sequences using the Micron Automata Processor," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 13, no. 1, pp. 99–111, 2016.

[66] C. Bo, V. Dang, E. Sadredini, and K. Skadron, "Searching for potential gRNA off-target sites for CRISPR/Cas9 using automata processing across different platforms," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (USA), pp. 737–748, IEEE, 2018.

[67] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, "Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications," in *Lectures on Runtime Verification: Introductory and Advanced Topics* (E. Bartocci and Y. Falcone, eds.), vol. 10457 of *Lecture Notes in Computer Science*, pp. 135–175, Cham: Springer, 2018.

[68] J. C. Davis, "Rethinking regex engines to address ReDoS," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, (New York, NY, USA), pp. 1256–1258, ACM, 2019.

[69] A. R. Meyer and M. J. Fischer, "Economy of description by automata, grammars, and formal systems," in *2013 IEEE 54th Annual Symposium on Foundations*

of Computer Science, (Los Alamitos, CA, USA), pp. 188–191, IEEE Computer Society, 1971.

[70] J. Wadden, V. Dang, N. Brunelle, T. Tracy II, D. Guo, E. Sadredini, K. Wang, C. Bo, G. Robins, M. Stan, and K. Skadron, "ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, IEEE, 2016.

[71] M. Lenjani and M. R. Hashemi, "Tree-based scheme for reducing shared cache miss rate leveraging regional, statistical and temporal similarities," *IET Computers & Digital Techniques*, vol. 8, no. 1, pp. 30–48, 2014.

[72] T. Liu, Y. Yang, Y. Liu, Y. Sun, and L. Guo, "An efficient regular expressions compression algorithm from a new perspective," in *2011 Proceedings IEEE INFOCOM*, (New York, NY, USA), pp. 2129–2137, IEEE, Apr. 2011.

[73] R. Rahimi, E. Sadredini, M. Stan, and K. Skadron, "Grapefruit: An Open-Source, Full-Stack, and Customizable Automata Processing on FPGAs," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, (New York, NY, USA), pp. 138–147, IEEE, May 2020.

[74] T. Xie, V. Dang, J. Wadden, K. Skadron, and M. Stan, "REAPR: Reconfigurable engine for automata processing," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, (New York, NY, USA), pp. 1–8, IEEE, 2017.

[75] P. Tandon, F. M. Sleiman, M. J. Cafarella, and T. F. Wenisch, "HAWK: Hardware support for unstructured log processing," in *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, (New York, NY, USA), pp. 469–480, IEEE, 2016.

[76] J. V. Lunteren, C. Hagleitner, T. Heil, G. Biran, U. Shvadron, and K. Atasu, "Designing a programmable wire-speed regular-expression matching accelerator," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, (New York, NY, USA), pp. 461–472, IEEE, 2012.

[77] K. Wang, K. Angstadt, C. Bo, N. Brunelle, E. Sadredini, T. Tracy, J. Wadden, M. Stan, and K. Skadron, "An overview of Micron's automata processor," in *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES '16, (New York, NY, USA), ACM, 2016.

[78] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, vol. 11, no. 6, pp. 419–422, 1968.

[79] V. M. Glushkov, "The abstract theory of automata," *Russian Math. Surveys*, vol. 16, no. 5, pp. 1–53, 1961.

[80] "TAQ database." Available at `https://www.nyse.com/`, 2023. [Online; Accessed 30 June, 2023].

[81] W. Zong, T. Heldt, G. B. Moody, and R. G. Mark, "An open-source algorithm to detect onset of arterial blood pressure pulses," in *Computers in Cardiology, 2003*, (New York, NY, USA), pp. 259–262, IEEE, Sep. 2003.

[82] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 439–455, ACM, 2013.

[83] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink: Stream and batch processing in a single engine," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, vol. 36, no. 4, pp. 28–38, 2015.

[84] "Apache Beam: An advanced unified programming model." Available at `https://beam.apache.org/`, 2023. [Online; Accessed 29 June 2023].

[85] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on Computers*, vol. C-36, pp. 24–35, Jan 1987.

[86] A. Benveniste, P. Le Guernic, and C. Jacquemot, "Synchronous programming with events and relations: The SIGNAL language and its semantics," *Science of Computer Programming*, vol. 16, no. 2, pp. 103–149, 1991.

[87] L. Kong and K. Mamouras, "StreamQL: A query language for processing streaming time series," in *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '20, (New York, NY, USA), pp. 183:1–183:32, Association for Computing Machinery, 2020.

[88] L. Kong, Q. Yu, A. Chattopadhyay, A. Le Glaunec, Y. Huang, K. Mamouras, and K. Yang, "Software-hardware codesign for efficient in-memory regular pattern

matching," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, (New York, NY, USA), pp. 733–748, ACM, 2022.

[89] A. Le Glaunec, L. Kong, and K. Mamouras, "Regular expression matching using bit vector automata," *Proc. ACM Program. Lang.*, vol. 7, apr 2023.

[90] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "STREAM: The Stanford data stream management system," in *Data Stream Management: Processing High-Speed Data Streams* (M. Garofalakis, J. Gehrke, and R. Rastogi, eds.), pp. 317–336, Berlin, Heidelberg: Springer, 2016.

[91] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman, "Continuously adaptive continuous queries over streams," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, (New York, NY, USA), pp. 49–60, ACM, 2002.

[92] J. F. Naughton, D. J. DeWitt, D. Maier, A. Aboulnaga, J. Chen, L. Galanis, J. Kang, R. Krishnamurthy, Q. Luo, N. Prakash, *et al.*, "The Niagara Internet query system," *IEEE Data Engineering Bulletin*, 2001.

[93] M. A. Hammad, M. F. Mokbel, M. H. Ali, W. G. Aref, A. C. Catlin, A. K. Elmagarmid, M. Eltabakh, M. G. Elfeky, T. M. Ghanem, R. Gwadera, I. F. Ilyas, M. Marzouk, and X. Xiong, "Nile: A query processing engine for data streams," in *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, (New York, NY, USA), pp. 851–851, IEEE, April 2004.

[94] R. S. Barga, J. Goldstein, M. H. Ali, and M. Hong, "Consistent streaming

through time: A vision for event stream processing," in *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, (Asilomar, CA, USA), pp. 363–374, www.cidrdb.org, 2007.

[95] M. H. Ali, C. Gerea, B. S. Raman, B. Sezgin, T. Tarnavski, T. Verona, P. Wang, P. Zabback, A. Ananthanarayan, A. Kirilov, M. Lu, A. Raizman, R. Krishnan, R. Schindlauer, T. Grabs, S. Bjeletich, B. Chandramouli, J. Goldstein, S. Bhat, Y. Li, V. Di Nicola, X. Wang, D. Maier, S. Grell, O. Nano, and I. Santos, "Microsoft CEP Server and online behavioral targeting," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1558–1561, 2009.

[96] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, (New York, NY, USA), pp. 170–177, IEEE, 2010.

[97] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce online," in *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, (USA), p. 21, USENIX Association, 2010.

[98] O. Boykin, S. Ritchie, I. O'Connell, and J. Lin, "Summingbird: A framework for integrating batch and online MapReduce computations," *Proceedings of the VLDB Endowment*, vol. 7, pp. 1441–1451, Aug. 2014.

[99] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Presented as part of*

the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12), (San Jose, CA), pp. 15–28, USENIX, 2012.

[100] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle, "MillWheel: Fault-tolerant stream processing at Internet scale," *Proceedings of the VLDB Endowment*, vol. 6, no. 11, pp. 1033–1044, 2013.

[101] S. A. Noghabi, K. Paramasivam, Y. Pan, N. Ramesh, J. Bringhurst, I. Gupta, and R. H. Campbell, "Samza: Stateful scalable stream processing at LinkedIn," *Proceedings of the VLDB Endowment*, vol. 10, no. 12, pp. 1634–1645, 2017.

[102] Y. Mei and S. Madden, "ZStream: A cost-based query processor for adaptively detecting composite events," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, (New York, NY, USA), pp. 193–206, ACM, 2009.

[103] A. Artikis, M. Sergot, and G. Paliouras, "An event calculus for event recognition," *IEEE Transactions on Knowledge and Data Engineering*, vol. 27, no. 4, pp. 895–908, 2015.

[104] I. Kolchinsky and A. Schuster, "Efficient adaptive detection of complex event patterns," *Proc. VLDB Endow.*, vol. 11, p. 1346–1359, jul 2018.

[105] M. Liu, E. Rundensteiner, K. Greenfield, C. Gupta, S. Wang, I. Ari, and A. Mehta, "E-Cube: Multi-dimensional event sequence analysis using hierarchical pattern query sharing," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), p. 889–900, Association for Computing Machinery, 2011.

[106] B. Chandramouli, J. Goldstein, M. Barnett, R. DeLine, D. Fisher, J. C. Platt, J. F. Terwilliger, and J. Wernsing, "Trill: A high-performance incremental query processor for diverse analytics," *Proceedings of the VLDB Endowment*, vol. 8, no. 4, pp. 401–412, 2014.

[107] EsperTech, "Esper." Available at `https://github.com/espertechinc/esper`, 2023. [Online; Accessed 29 March, 2023].

[108] S. Suhothayan, K. Gajasinghe, I. Loku Narangoda, S. Chaturanga, S. Perera, and V. Nanayakkara, "Siddhi: A second look at complex event processing architectures," in *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, (New York, NY, USA), pp. 43–50, ACM, 2011.

[109] Oracle, "Oracle stream analytics." Available at `https://www.oracle.com/middleware/technologies/stream-processing.html`, 2023. [Online; Accessed 30 June, 2023].

[110] B. Chandramouli, J. Goldstein, and Y. Li, "Impatience is a virtue: Revisiting disorder in high-performance log analytics," in *Proceedings of the IEEE 34th International Conference on Data Engineering*, ICDE 2018, pp. 677–688, IEEE, 2018.

[111] M. Nikolic, B. Chandramouli, and J. Goldstein, "Enabling signal processing over data streams," in *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, (New York, NY, USA), pp. 95–108, ACM, 2017.

[112] Oracle, "Java Stream." Available at `https://docs.oracle.com/javase/8/`, 2023. [Online; Accessed 1 July, 2023].

[113] O. Kiselyov, A. Biboudis, N. Palladinos, and Y. Smaragdakis, "Stream fusion, to completeness," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, (New York, NY, USA), pp. 285–299, ACM, 2017.

[114] Y. Yuan, D. Lin, A. Mishra, S. Marwaha, R. Alur, and B. T. Loo, "Quantitative network monitoring with NetQRE," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, (New York, NY, USA), p. 99–112, Association for Computing Machinery, 2017.

[115] K. Mamouras, M. Raghothaman, R. Alur, Z. G. Ives, and S. Khanna, "StreamQRE: Modular specification and efficient evaluation of quantitative queries over streaming data," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '17, (New York, NY, USA), pp. 693–708, ACM, 2017.

[116] R. Alur and K. Mamouras, "An introduction to the StreamQRE language," *Dependable Software Systems Engineering*, vol. 50, pp. 1–24, 2017.

[117] R. Alur, K. Mamouras, and C. Stanford, "Modular quantitative monitoring," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 50:1–50:31, 2019.

[118] R. Alur, D. Fisman, K. Mamouras, M. Raghothaman, and C. Stanford, "Streamable regular transductions," *Theoretical Computer Science*, vol. 807, pp. 15–41, 2020.

[119] R. Alur, K. Mamouras, and C. Stanford, "Automata-based stream processing," in *Proceedings of the 44th International Colloquium on Automata, Languages,*

and Programming (ICALP '17) (I. Chatzigiannakis, P. Indyk, F. Kuhn, and A. Muscholl, eds.), vol. 80 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 112:1–112:15, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017.

[120] R. Alur, K. Mamouras, and D. Ulus, "Derivatives of quantitative regular expressions," in *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday* (L. Aceto, G. Bacci, G. Bacci, A. Ingólfsdóttir, A. Legay, and R. Mardare, eds.), vol. 10460 of *Lecture Notes in Computer Science*, pp. 75–95, Cham: Springer, 2017.

[121] "InfluxDB: The time series data platform where developers build IoT, analytics, and cloud applications.." Available at `https://www.influxdata.com/`, 2023. [Online; Accessed 29 June 2023].

[122] "Query and code together with Flux." Available at `https://www.influxdata.com/products/flux/`, 2023. [Online; Accessed 29 June 2023].

[123] E. Bainomugisha, A. L. Carreton, T. van Cutsem, S. Mostinckx, and W. de Meuter, "A survey on reactive programming," *ACM Computing Surveys*, vol. 45, pp. 52:1–52:34, Aug. 2013.

[124] H. Nilsson, J. Peterson, and P. Hudak, "Functional hybrid modeling," in *Practical Aspects of Declarative Languages*, (Berlin, Heidelberg), pp. 376–390, Springer Berlin Heidelberg, 2003.

[125] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi, "Flapjax: A programming language for Ajax applications," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented*

Programming Systems Languages and Applications, OOPSLA '09, (New York, NY, USA), pp. 1–20, ACM, 2009.

[126] E. Czaplicki and S. Chong, "Asynchronous functional reactive programming for GUIs," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), pp. 411–422, ACM, 2013.

[127] "Reactivex." Available at `http://reactivex.io/`, 2023. [Online; Accessed 29 June, 2023].

[128] VMware, "Project Reactor: Create efficient reactive systems." Available at `https://projectreactor.io/`, 2023. [Online; Accessed 29 June 2023].

[129] Lightbend, "Akka streams." Available at `https://akka.io/`, 2020. [Online; Accessed 10 June, 2020].

[130] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "The case for a signal-oriented data stream management system," in *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research (CIDR '07)*, pp. 397–406, 2007.

[131] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, and S. Madden, "XStream: a signal-oriented data stream management system," in *2008 IEEE 24th International Conference on Data Engineering*, (New York, NY, USA), pp. 1180–1189, IEEE, April 2008.

[132] A. Jayarajan, K. Hau, A. Goodwin, and G. Pekhimenko, "LifeStream: A high-performance stream processing engine for periodic streams," in *Proceedings of the*

26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2021, (New York, NY, USA), p. 107–122, Association for Computing Machinery, 2021.

[133] Snort, "Snort - network intrusion detection & prevention system." Available at `https://www.snort.org/`, 2023. [Online; Accessed 30 June, 2023].

[134] Suricata, "Suricata - open source intrusion detection and prevention engine." Available at `https://suricata.io/`, 2023. [Online; Accessed 30 June, 2023].

[135] C. J. A. Sigrist, L. Cerutti, E. de Castro, P. S. Langendijk-Genevaux, V. Bulliard, A. Bairoch, and N. Hulo, "PROSITE, a protein domain database for functional characterization and annotation," *Nucleic Acids Research*, vol. 38, no. suppl_1, pp. D161–D166, 2009.

[136] "Look Around in PCRE." Available at `https://www.pcre.org/original/doc/html/pcrepattern.html#SEC20`, 2023. [Online; Accessed 30 June, 2023].

[137] "Back reference in PCRE." Available at `https://www.pcre.org/original/doc/html/pcrepattern.html#SEC19`, 2023. [Online; Accessed 30 June, 2023].

[138] J. Bispo, I. Sourdis, J. M. P. Cardoso, and S. Vassiliadis, "Regular expression matching for reconfigurable packet inspection," in *2006 IEEE International Conference on Field Programmable Technology*, (USA), pp. 119–126, IEEE, 2006.

[139] Y.-H. E. Yang, W. Jiang, and V. K. Prasanna, "Compact architecture for high-throughput regular expression matching on FPGA," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, (New York, NY, USA), p. 30–39, ACM, 2008.

[140] I. Sourdis, J. Bispo, J. M. Cardoso, and S. Vassiliadis, "Regular expression matching in reconfigurable hardware," *Journal of Signal Processing Systems*, vol. 51, no. 1, pp. 99–121, 2008.

[141] J. Chen, X. Zhang, T. Wang, Y. Zhang, T. Chen, J. Chen, M. Xie, and Q. Liu, "Fidas: Fortifying the cloud via comprehensive FPGA-based offloading for intrusion detection: Industrial product," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, (New York, NY, USA), p. 1029–1041, Association for Computing Machinery, 2022.

[142] Z. Zhao, H. Sadok, N. Atre, J. C. Hoe, V. Sekar, and J. Sherry, "Achieving 100gbps intrusion prevention on a single server," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 1083–1100, USENIX Association, Nov. 2020.

[143] M. Ceška, V. Havlena, L. Holík, J. Korenek, O. Lengál, D. Matoušek, J. Matoušek, J. Semric, and T. Vojnar, "Deep packet inspection in FPGAs via approximate nondeterministic automata," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 109–117, IEEE, 2019.

[144] M. Becchi and P. Crowley, "Extending finite automata to efficiently match Perl-compatible regular expressions," in *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, (New York, NY, USA), ACM, 2008.

[145] Y.-H. E. Yang and V. K. Prasanna, "Space-time tradeoff in regular expression matching with semi-deterministic finite automata," in *2011 Proceedings IEEE INFOCOM*, (New York, NY, USA), pp. 1853–1861, IEEE, 2011.

[146] H. Nakahara, T. Sasao, and M. Matsuura, "A regular expression matching circuit based on a decomposed automaton," in *Reconfigurable Computing: Architectures, Tools and Applications*, (Heidelberg), pp. 16–28, Springer, 2011.

[147] R. A. Baeza-Yates and G. H. Gonnet, "Efficient text searching of regular expressions," in *Automata, Languages and Programming* (G. Ausiello, M. Dezani-Ciancaglini, and S. R. Della Rocca, eds.), (Heidelberg), pp. 46–62, Springer, 1989.

[148] H. Liu, M. Ibrahim, O. Kayiran, S. Pai, and A. Jog, "Architectural support for efficient large-scale automata processing," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, (New York, NY, USA), pp. 908–920, IEEE, 2018.

[149] J. Wadden, T. Tracy, E. Sadredini, L. Wu, C. Bo, J. Du, Y. Wei, J. Udall, M. Wallace, M. Stan, and K. Skadron, "AutomataZoo: A modern automata processing benchmark suite," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, (New York, NY, USA), pp. 13–24, IEEE, 2018.

[150] I. Roy, A. Srivastava, M. Grimm, M. Nourian, M. Becchi, and S. Aluru, "Evaluating high performance pattern matching on the automata processor," *IEEE Transactions on Computers*, vol. 68, no. 8, pp. 1201–1212, 2019.

[151] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *Commun. ACM*, vol. 18, p. 333–340, jun 1975.

[152] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch, "HARE: Hardware accelerator for regular expressions," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[153] N. Cascarano, P. Rolando, F. Risso, and R. Sisto, "iNFAnt: NFA pattern matching on GPGPU devices," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 5, pp. 20–26, 2010.

[154] Y. Zu, M. Yang, Z. Xu, L. Wang, X. Tian, K. Peng, and Q. Dong, "GPU-based NFA implementation for memory efficient high speed regular expression matching," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), p. 129–140, ACM, 2012.

[155] H. Liu, S. Pai, and A. Jog, "Asynchronous automata processing on GPUs," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 7, mar 2023.

[156] Y. Wang, R. Watling, J. Qiu, and Z. Wang, "GSpecPal: Speculation-centric finite state machine parallelization on GPUs," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, (New York, NY, USA), pp. 481–491, IEEE, 2022.

[157] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, (USA), p. 187–201, IEEE Computer Society, 2008.

[158] L. Holík, O. Lengál, O. Saarikivi, L. Turoňová, M. Veanes, and T. Vojnar, "Succinct determinisation of counting automata via sphere construction," in *APLAS 2019* (A. W. Lin, ed.), vol. 11893 of *Lecture Notes in Computer Science*, (Cham), pp. 468–489, Springer, 2019.

[159] L. Turoňová, L. Holík, O. Lengál, O. Saarikivi, M. Veanes, and T. Vojnar, "Regex matching with counting-set automata," *Proceedings of the ACM on Programming*

Languages, vol. 4, no. OOPSLA, 2020.

[160] L. Dagum and R. Menon, "OpenMP: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[161] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, (New York, NY, USA), p. 79–90, Association for Computing Machinery, 2009.

[162] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), p. 207–216, Association for Computing Machinery, 1995.

[163] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, (New York, NY, USA), p. 519–538, Association for Computing Machinery, 2005.

[164] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[165] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, p. 223–252, sep 1977.

[166] C. A. Petri, *Kommunikation mit automaten*. PhD thesis, University of Bonn, West Germany, 1962.

[167] G. Estrin and R. Turn, "Automatic assignment of computations in a variable structure computer system," *IEEE Transactions on Electronic Computers*, vol. EC-12, no. 6, pp. 755–773, 1963.

[168] R. M. Karp and R. E. Miller, "Properties of a model for parallel computations: Determinacy, termination, queueing," *SIAM Journal on Applied Mathematics*, vol. 14, no. 6, pp. 1390–1411, 1966.

[169] G. Kahn, "The semantics of a simple language for parallel programming," *Information Processing*, vol. 74, pp. 471–475, 1974.

[170] T. Goubier, R. Sirdey, S. Louise, and V. David, "ΣC: A programming model and language for embedded manycores," in *Algorithms and Architectures for Parallel Processing*, (Berlin, Heidelberg), pp. 385–394, Springer Berlin Heidelberg, 2011.

[171] G. Agha, *Actors: a model of concurrent computation in distributed systems*. MIT press, 1986.

[172] F. D. Boer, V. Serbanescu, R. Hähnle, L. Henrio, J. Rochas, C. C. Din, E. B. Johnsen, M. Sirjani, E. Khamespanah, K. Fernandez-Reyes, and A. M. Yang, "A survey of active object languages," *ACM Comput. Surv.*, vol. 50, oct 2017.

[173] J. Armstrong, *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2013.

[174] M. Sirjani, A. Movaghar, A. Shali, and F. S. de Boer, "Modeling and verification of reactive systems using Rebeca," *Fundamenta Informaticae*, vol. vol. 63, no. 4,

pp. 385–410, 2004.

[175] M. Sirjani, "Rebeca: Theory, applications, and tools," in *Formal Methods for Components and Objects*, (Berlin, Heidelberg), pp. 102–126, Springer Berlin Heidelberg, 2007.

[176] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen, "ABS: A core language for abstract behavioral specification," in *Formal Methods for Components and Objects*, (Berlin, Heidelberg), pp. 142–164, Springer Berlin Heidelberg, 2012.

[177] D. Caromel and L. Henrio, *A Theory of Distributed Objects*. Springer, 2005.

[178] S. Brandauer, E. Castegren, D. Clarke, K. Fernandez-Reyes, E. B. Johnsen, K. I. Pun, S. L. T. Tarifa, T. Wrigstad, and A. M. Yang, *Parallel Objects for Multicores: A Glimpse at the Parallel Language Encore*, pp. 1–56. Cham: Springer International Publishing, 2015.

[179] S. Tasharofi, M. Pradel, Y. Lin, and R. Johnson, "Bita: Coverage-guided, automatic testing of actor programs," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 114–124, 2013.

[180] K. Shibanai and T. Watanabe, "Actoverse: A reversible debugger for actors," in *Proceedings of the 7th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, AGERE 2017, (New York, NY, USA), p. 50–57, Association for Computing Machinery, 2017.

[181] A. S. Mathur, B. K. Ozkan, and R. Majumdar, "Idea: An immersive debugger for actors," in *Proceedings of the 17th ACM SIGPLAN International Workshop on*

Erlang, Erlang 2018, (New York, NY, USA), p. 1–12, Association for Computing Machinery, 2018.

[182] C. T. Lopez, R. G. Singh, S. Marr, E. G. Boix, and C. Scholliers, "Multiverse Debugging: Non-deterministic debugging for non-deterministic programs," in *33rd European Conference on Object-Oriented Programming*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, April 2019.

[183] C. Menard, M. Lohstroh, S. Bateni, M. Chorlian, A. Deng, P. Donovan, C. Fournier, S. Lin, F. Suchert, T. Tanneberger, H. Kim, J. Castrillon, and E. A. Lee, "High-performance deterministic concurrency using Lingua Franca," 2023.

[184] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1989.

[185] M. Vanneschi, "The programming model of assist, an environment for parallel and distributed portable applications," *Parallel Computing*, vol. 28, no. 12, pp. 1709–1732, 2002.

[186] D. Caromel and M. Leyton, "Fine tuning algorithmic skeletons," in *Euro-Par 2007 Parallel Processing*, (Berlin, Heidelberg), pp. 72–81, Springer Berlin Heidelberg, 2007.

[187] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí, "Parallel functional programming in Eden," *Journal of Functional Programming*, vol. 15, no. 3, pp. 431–475, 2005.

[188] M. Aldinucci, M. Danelutto, and P. Dazzi, "Muskel: an expandable skeleton environment," *Scalable Computing: Practice and Experience*, vol. 8, no. 4, 2007.

[189] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "p^3l: A structured high-level parallel language, and its structured support," *Concurrency: Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[190] M. Leyton and J. M. Piquer, "Skandium: Multi-core programming with algorithmic skeletons," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp. 289–296, 2010.

[191] H. Miao, H. Park, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox: Modern stream processing on a multicore machine," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 617–629, USENIX Association, July 2017.

[192] H. Miao, M. Jeon, G. Pekhimenko, K. S. McKinley, and F. X. Lin, "StreamBox-HBM: Stream analytics on high bandwidth hybrid memory," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 167–181, Association for Computing Machinery, 2019.

[193] H. Park, S. Zhai, L. Lu, and F. X. Lin, "StreamBox-TZ: Secure stream analytics at the edge with TrustZone," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, (Renton, WA), pp. 537–554, USENIX Association, July 2019.

[194] Microsoft, "PLINQ: Parallel implementation of the Language-Integrated Query (LINQ) pattern." Available at `https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq`, 2023. [Online; Accessed 30 June, 2023].

[195] C. Csallner, L. Fegaras, and C. Li, "New ideas track: Testing Mapreduce-style programs," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, (New York, NY, USA), p. 504–507, Association for Computing Machinery, 2011.

[196] Y.-F. Chen, L. Song, and Z. Wu, "The commutativity problem of the MapReduce framework: A transducer-based approach," in *Computer Aided Verification*, (Cham), pp. 91–111, Springer International Publishing, 2016.

[197] Z. Xu, M. Hirzel, and G. Rothermel, "Semantic characterization of MapReduce workloads," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, (New York, USA), pp. 87–97, IEEE, 2013.

[198] Z. Xu, M. Hirzel, G. Rothermel, and K. Wu, "Testing properties of dataflow program operators," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (New York, USA), pp. 103–113, IEEE, 2013.

[199] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "DiffStream: Differential output testing for stream processing programs," *Proc. ACM Program. Lang.*, vol. 4, Nov. 2020.

[200] K. Mamouras, C. Stanford, R. Alur, Z. G. Ives, and V. Tannen, "Data-trace types for distributed stream processing systems," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, (New York, NY, USA), pp. 670–685, ACM, 2019.

[201] R. Alur, P. Hilliard, Z. G. Ives, K. Kallas, K. Mamouras, F. Niksic, C. Stanford, V. Tannen, and A. Xue, "Synchronization schemas," in *Proceedings of the 40th*

ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2021, (New York, NY, USA), pp. 1–18, ACM, 2021.

[202] K. Kallas, F. Niksic, C. Stanford, and R. Alur, "Stream processing with dependency-guided synchronization," in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '22, (New York, NY, USA), p. 1–16, Association for Computing Machinery, 2022.

[203] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, (New York, NY, USA), p. 212–223, Association for Computing Machinery, 1998.

[204] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, (New York, NY, USA), p. 36–43, Association for Computing Machinery, 2000.

[205] "RxJava: Reactive extensions for the JVM." Available at `https://github.com/ReactiveX/RxJava`, 2023. [Online; Accessed 30 June, 2023].

[206] R. Alur and P. Černý, "Expressiveness of streaming string transducers," in *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2010)*, vol. 8 of *Leibniz International Proceedings in Informatics (LIPIcs)*, (Dagstuhl, Germany), pp. 1–12, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[207] J. Pan and W. J. Tompkins, "A real-time QRS detection algorithm," *IEEE Transactions on Biomedical Engineering*, vol. BME-32, pp. 230–236, March 1985.

[208] R. O. Bert-Uwe Köhler, Carsten Hennig, "The principles of software QRS detection," *IEEE Engineering in Medicine and Biology Magazine*, vol. 21, pp. 42–57, Jan 2002.

[209] G. B. Moody, "Single-channel QRS detector." Available at `https://www.physionet.org/physiotools/wag/sqrs-1.htm`, 2023. [Online; Accessed 29 June 2023].

[210] R. Alur, K. Mamouras, C. Stanford, and V. Tannen, "Interfaces for stream processing systems," in *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday* (M. Lohstroh, P. Derler, and M. Sirjani, eds.), vol. 10760 of *Lecture Notes in Computer Science*, pp. 38–60, Cham: Springer, 2018.

[211] K. Mamouras and Z. Wang, "Online signal monitoring with bounded lag," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 11, pp. 3868–3880, 2020.

[212] K. Mamouras, "Semantic foundations for deterministic dataflow and stream processing," in *Proceedings of the 29th European Symposium on Programming (ESOP '20)* (P. Müller, ed.), vol. 12075 of *Lecture Notes in Computer Science*, (Berlin, Heidelberg), pp. 394–427, Springer, 2020.

[213] P. Tucker, K. Tufte, V. Papadimos, and D. Maier, "A benchmark for queries over data streams." Available at `http://datalab.cs.pdx.edu/niagara/NEXMark/`, 2002. [Online; Accessed 30 June, 2023].

[214] "Trill Documentation: best practices for using Trill in real-time deployments." Available at `https://github.com/microsoft/Trill/blob/master/`

Documentation/BestPractices.pdf, 2018. [Online; Accessed 30 June, 2023].

[215] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Rec.*, vol. 34, pp. 39–44, Mar. 2005.

[216] K. Tangwongsan, M. Hirzel, S. Schneider, and K.-L. Wu, "General incremental sliding-window aggregation," *Proc. VLDB Endow.*, vol. 8, pp. 702–713, Feb. 2015.

[217] M. Hirzel, S. Schneider, and K. Tangwongsan, "Sliding-window aggregation algorithms: Tutorial," in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, (New York, NY, USA), pp. 11–14, ACM, 2017.

[218] M. Datar, A. Gionis, P. Indyk, and R. Motwani, "Maintaining stream statistics over sliding windows," *SIAM Journal on Computing*, vol. 31, no. 6, pp. 1794–1813, 2002.

[219] A. Arasu and J. Widom, "Resource sharing in continuous sliding-window aggregates," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, (Toronto, Canada), p. 336–347, VLDB Endowment, 2004.

[220] "SASE: Open source system." Available at https://github.com/haopeng/sase, 2014. [Online; Accessed 30 June 2023].

[221] L. Brenna, A. Demers, J. Gehrke, M. Hong, J. Ossher, B. Panda, M. Riedewald, M. Thatte, and W. White, "Cayuga: A high-performance event processing

engine," in *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, (New York, NY, USA), p. 1100–1102, Association for Computing Machinery, 2007.

[222] M. F. O'Rourke, "The arterial pulse in health and disease," *American Heart Journal*, vol. 82, no. 5, pp. 687 – 702, 1971.

[223] H. Abbas, A. Rodionova, K. Mamouras, E. Bartocci, S. A. Smolka, and R. Grosu, "Quantitative regular expressions for arrhythmia detection," *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, vol. 16, no. 5, pp. 1586–1597, 2019.

[224] H. Abbas, R. Alur, K. Mamouras, R. Mangharam, and A. Rodionova, "Real-time decision policies with predictable performance," *Proceedings of the IEEE, Special Issue on Design Automation for Cyber-Physical Systems*, vol. 106, no. 9, pp. 1593–1615, 2018.

[225] A. Subramaniyan, J. Wang, E. R. M. Balasubramanian, D. Blaauw, D. Sylvester, and R. Das, "Cache automaton," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, (New York, NY, USA), p. 259–272, ACM, 2017.

[226] E. Sadredini, R. Rahimi, M. Lenjani, M. Stan, and K. Skadron, "Impala: Algorithm/Architecture Co-Design for In-Memory Multi-Stride Pattern Matching," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, (New York, NY, USA), pp. 86–98, IEEE, Feb. 2020.

[227] L. J. Stockmeyer and A. R. Meyer, "Word problems requiring exponential time (preliminary report)," in *Proceedings of the Fifth Annual ACM Symposium on*

Theory of Computing, STOC '73, (New York, NY, USA), pp. 1–9, ACM, 1973.

[228] K. Angstadt, J. Wadden, W. Weimer, and K. Skadron, "MNRL and MNCaRT: An open-source, multi-architecture state machine research and execution ecosystem," Tech. Rep. CS2017-01, University of Virginia, 2017.

[229] A. S. Foundation, "Apache SpamAssassin." Available at `https://spamassassin.apache.org/`, 2022. [Online; Accessed 30 June, 2023].

[230] W. Gelade, M. Gyssens, and W. Martens, "Regular expressions with counting: Weak versus strong determinism," in *Mathematical Foundations of Computer Science 2009*, (Heidelberg), pp. 369–381, Springer, 2009.

[231] A. R. Meyer and L. J. Stockmeyer, "The equivalence problem for regular expressions with squaring requires exponential space," in *13th Annual Symposium on Switching and Automata Theory (SWAT 1972)*, (Los Alamitos, CA, USA), pp. 125–129, IEEE Computer Society, 1972.

[232] K. Mamouras, A. Chattopadhyay, and Z. Wang, "Algebraic quantitative semantics for efficient online temporal monitoring," in *TACAS 2021* (J. F. Groote and K. G. Larsen, eds.), vol. 12651 of *Lecture Notes in Computer Science*, (Cham), pp. 330–348, Springer, 2021.

[233] K. Mamouras, A. Chattopadhyay, and Z. Wang, "A compositional framework for quantitative online monitoring over continuous-time signals," in *RV 2021* (L. Feng and D. Fisman, eds.), vol. 12974 of *Lecture Notes in Computer Science*, (Cham), pp. 142–163, Springer, 2021.

[234] ClamAV, "ClamAV - open source antivirus engine." Available at `https://www.clamav.net/`, 2023. [Online; Accessed 30 June, 2023].

[235] E. Sadredini, R. Rahimi, V. Verma, M. Stan, and K. Skadron, "eAP: A scalable and efficient in-memory accelerator for automata processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, (New York, NY, USA), pp. 87—-99, ACM, 2019.

[236] Materialize, "Materialize: The streaming database you already know how to use." Available at `https://materialize.com/`, 2023. [Online; Accessed 29 June 2023].

[237] D. G. Murray, F. McSherry, M. Isard, R. Isaacs, P. Barham, and M. Abadi, "Incremental, iterative data processing with timely dataflow," *Commun. ACM*, vol. 59, p. 75–83, sep 2016.

[238] "Fearless concurrency." Available at `https://doc.rust-lang.org/book/ch16-00-concurrency.html`, 2023. [Online; Accessed 10 Aug, 2023].

[239] F. McSherry, A. Lattuada, M. Hoffmann, and N. Benesch, "Timely Dataflow." Available at `https://github.com/TimelyDataflow/timely-dataflow`, 2023. [Online; Accessed 29 Jun, 2023].

[240] P. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003.

[241] Z. Qian, Y. He, C. Su, Z. Wu, H. Zhu, T. Zhang, L. Zhou, Y. Yu, and Z. Zhang, "TimeStream: Reliable stream computation in the cloud," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), p. 1–14, Association for Computing Machinery, 2013.

[242] "Personalized web search challenge." Available at `https://www.kaggle.com/c/yandex-personalized-web-search-challenge`, 2014. [Online; Accessed 30 June, 2023].

[243] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, Z.-W. Lin, and V. Kurien, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, (New York, NY, USA), p. 139–152, Association for Computing Machinery, 2015.

[244] "DEBS 2014 grand challenge: Smart Homes." Available at `https://debs.org/grand-challenges/2014/`, 2014. [Online; Accessed 30 June 2023].

[245] "Relative Strength Index." Available at `https://en.wikipedia.org/wiki/Relative_strength_index`, 2023. [Online; Accessed 30 June 2023].

[246] A. Ţăran-Moroşan, "The relative strength index revisited," *African Journal of Business Management*, vol. 5, no. 14, pp. 5855–5862, 2011.

[247] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[248] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[249] D. Anguita, A. Ghio, L. Oneto, X. Parra Perez, and J. L. Reyes Ortiz, "A public domain dataset for human activity recognition using smartphones," in

Proceedings of the 21th International European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, pp. 437–442, 2013.

[250] A. Khadersab and S. Shivakumar, "Vibration analysis techniques for rotating machinery and its effect on bearing faults," *Procedia Manufacturing*, vol. 20, pp. 247–252, 2018. 2nd International Conference on Materials, Manufacturing and Design Engineering (iCMMD2017), 11-12 December 2017, MIT Aurangabad, Maharashtra, INDIA.

[251] "Kurtosis." Available at `https://en.wikipedia.org/wiki/Kurtosis`, 2023. [Online; Accessed 30 June 2023].

[252] "Crest Factor." Available at `https://en.wikipedia.org/wiki/Crest_factor`, 2023. [Online; Accessed 30 June 2023].

[253] H. Huang and N. Baddour, "Bearing vibration data collected under time-varying rotational speed conditions," *Data in Brief*, vol. 21, pp. 1745–1749, 2018.

[254] "Fraud detection with Flink." Available at `https://nightlies.apache.org/flink/flink-docs-master/docs/try-flink/datastream/`, Online; Accessed 30 June, 2023.

[255] R. Wu and E. J. Keogh, "Current time series anomaly detection benchmarks are flawed and are creating the illusion of progress," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 3, pp. 2421–2429, 2023.

[256] J. C. Beard, P. Li, and R. D. Chamberlain, "RaftLib: A C++ template library for high performance stream parallel processing," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores*

and Manycores, PMAM '15, (New York, NY, USA), p. 96–105, Association for Computing Machinery, 2015.