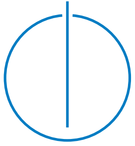
Mohammadreza Najafi

Hardware Accelerated Stream Processing

Technische
Universität
München



Technische Universität München

Fakultät für Informatik

Hardware Accelerated Stream Processing

Mohammadreza Najafi

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

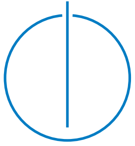
genehmigten Dissertation.

Vorsitzender: Prof. Dr. Hans Michael Gerndt

Prüfer der Dissertation:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. Charles Zhang

Die Dissertation wurde am 05.06.2019 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 21.10.2019 angenommen.



Technische Universität München

Department of Informatics

Hardware Accelerated Stream Processing

Mohammadreza Najafi

Complete copy of the dissertation approved by the Department of Informatics of the Technical University of Munich in partial fulfillment of the requirements for the degree of

Doktors der Naturwissenschaften (Dr. rer. nat.)

Chair: Prof. Dr. Hans Michael Gerndt

Dissertation examiners:

1. Prof. Dr. Hans-Arno Jacobsen
2. Prof. Dr. Charles Zhang

The dissertation was submitted to the Technical University of Munich on 05.06.2019 and accepted by the degree-awarding institution of Department of Informatics on 21.10.2019

Abstract

Exponential growth in data generation, emerged from an increase in the number of devices connected to the World Wide Web, and also an increase in the precision of data gathered pose a threat to data centers while opening new potential for data analysis when the processing is kept tractable. To deal with this ever growing and already enormous amount of data, hardware acceleration is receiving much attention due to its great efficiency and parallel capabilities. Hardware acceleration provides the best in class performance and power consumption properties; however, it is hindered by high research and development costs and long times-to-market that are repeated for each new application. In this perspective, by means of easier reconfigurability, better availability, and more convenient integration with the current infrastructures, field programmable gate arrays (FPGAs) are already finding their way into industrial data centers and commercial acceleration services.

We propose a reconfigurable hardware-based streaming architecture, namely flexible query processor (FQP), that constitutes a family of stream processing blocks that support dynamic changes to queries and streams, as well as static changes to the processor-internal fabric in order to maximize performance for given workloads. While processing incoming tuples, FQP can accept new queries, a key characteristic distinguishing FQP from related approaches. As an extension to FQP, we add the support for flexible data dimensions, in particular a segment-at-a-time mechanism, to realize processing of tuples of variable sizes. While many of these features are readily available in software, their hardware-based realizations have been one of the main shortcomings of existing research efforts.

FQP is designed to support the stream join parallelization, as one of the most resource-intensive operations. As a result, it utilizes a bidirectional data-flow model that was the only available stream join parallelization model at the time. This model complicated the design of FQP by forcing streams into two separate data-paths, imposing large controlling logics in each processing block to prevent race conditions and to ensure the correct execution. To address these fundamental issues, we introduce a novel unidirectional data-flow model for low-latency stream join processing parallelization, referred to as SplitJoin, that operates by splitting the join operation into independent storing and processing steps that gracefully scale with respect to the number of cores.

In the context of stream processing, it is common to have queries that use multiple data streams simultaneously. This marks the multiway stream joins operator as one of the essential blocks of our query processor. The challenges for multiway stream joins come from the required real-time join operator reordering when intermediate results are not materialized (due to their potentially large sizes). We propose a scalable circular pipeline design, namely Circular-MJ, which realizes the various necessary join trees using a fixed operator ordering and an arbitrary tuple input mechanism. Circular-MJ reduces the reordering challenge to an input reordering problem, which is later addressed by a pipeline distribution chain. We further propose an optimized pipeline stream join (Stashed-MJ) that uses a best-effort buffering technique to maintain intermediate results. Lastly, we present a parallelized version of our multiway stream join by integrating our proposed pipelines into a parallel

unidirectional flow-based architecture (Parallel-MJ).

As another crucial operator necessary for our query processor, we propose our novel hash-based stream join solution (HB-SJ) in hardware to accelerate processing of equi-joins, the most common type of join operators. HB-SJ utilizes two Murmur3 hash functions with four storage tables. In case the selected row is full in all four tables, the new tuple is inserted into an overflow buffer. To preserve the tuples' arrival ordering (necessary in the count-based sliding window), our solution uses an order sliding window. Choosing proper parameters for HB-SJ enables extremely fast (in the order of nanoseconds) sliding window lookups independent of the actual size of the window.

As the last part in this dissertation, we present our simplex stream processor (SSP), a successor to our custom flexible query processor that targets real-time stream processing while providing modular components. SSP benefits from our stream customized network-on-chip which uses a unidirectional data-flow model. By benefiting from our proposed solutions (i.e., SplitJoin, Stashed-MJ, Parallel-MJ, HB-SJ), SSP introduces libraries for the communication network and processing blocks, with a consistent interface that allows the further addition of components. As a proof of concept, we benchmark a modified version of the TPC-H third query on our SSP, realized in VHDL, while presenting the query mapping, programming, and processing steps in detail.

Zusammenfassung

Die exponentielle Zunahme an maschinell generierten Daten stellt Datencenter vor neue Herausforderungen. Dieser Trend wird getrieben durch eine steigende Anzahl an internetfähigen Geräten und der Erhebungsgenauigkeit von Sensoren. Dies ermöglicht aber auch neue Anwendungszenarien in der Datenanalyse. Um mit den steigenden Datenmengen umgehen zu können bekommt Hardwarebeschleunigung wegen ihrer Effizienz sowie Parallelisierungseigenschaften derzeit große Aufmerksamkeit. Field Programmable Gate Arrays (FPGAs) haben in diesem Zusammenhang wegen ihrer einfachen Rekonfigurierbarkeit, ihrer hohen Verfügbarkeit sowie der komfortablen Integration in existierende Infrastrukturen bereits Einzug in industrielle Datencenter und kommerzielle Beschleunigungsservices gefunden.

In dieser Arbeit wird eine rekonfigurierbare Hardware-basierte Streamingarchitektur, auch Flexible Query Processor (FQP) genannt, vorgestellt. FQP besteht aus einer Familie von Datenstromverarbeitungsblöcken. Diese Blöcke passen sich dynamisch an die Anfragen und Datenströme an. Weiters werden statische Anpassungen im prozessorinternen Gefüge zur Leistungssteigerung für bestimmte Arbeitslasten eingesetzt. Im Gegensatz zu anderen Ansätzen akzeptiert FQP neue Anfragen auch während der Verarbeitung eingehender Datensätze. Eine Erweiterung von FQP, der Segment-at-a-time Mechanismus, flexiblisiert die Datendimensionen wodurch Datensätze unterschiedlicher Größe verarbeitet werden können. Viele dieser Eigenschaften sind in Software verfügbar, eine Hardware-basierte Umsetzungen ist eine Lücke in der aktuellen Forschung.

Das Ziel von FQP ist Stream Joins, eine der ressourcenintensivsten Operationen, zu parallelisieren. FQP verwendet ein bidirektionales Datenflussmodell, welches das einzige zu dieser Zeit verfügbare Parallelisierungsmodell darstellte. Dieses Modell erschwerte das Design von FQP. Es erfordert aufwendig Kontrollogik da Streams in zwei separate Datenpfade aufgeteilt werden. Die Kontrollogik ist erforderlich um Race Conditions zu verhindern und die korrekte Ausführung zu garantieren. Wir stellen ein neues unidirektionales Datenflussmodell vor, genannt Stream Join Parallelisierung (SplitJoin), welches die Join Operationen in unabhängige Speicher- und Verarbeitungsschritte unterteilt, welche wiederum mit der Anzahl der Kerne skalieren.

Anfragen greifen auf mehrer Datenströme gleichzeitig zu. Der Multiway Stream Join ist ein wichtiger Verarbeitungsblock um diese Aufgabe zu erfüllen, hat aber siginifikante Skalierungsprobleme bei der Analyse von Datenströmen mit Hardwarebeschleunigung.

Die Herausforderungen bei Multiway Stream Join ist dabei die Neuanordnung der Join Operatoren in Echtzeit sofern Zwischenergebnisse nicht materialisiert werden. Die Arbeit stellt hierzu ein skalierbares ringförmiges Pipelinedesign mit Namen Circular-MJ vor, welches die verschiedenen notwendigen Join-Bäume durch eine starre Operatoranordnung sowie einen willkürlichen Datensätzen realisiert. Circular-MJ reduziert das Neuanordungsproblem der Operatoren auf das Eingabeordungsproblem, welches durch eine Pipeline an Verteiler gelöst wird. Darüber hinaus wird eine optimierte Pipeline Stream Join (Stashed-MJ) vorgestellt. Dieser verwendet einen best-effort Pufferansatz um

Zwischenergebnisse zu verwalten. Zuletzt wird eine parallelisierte Version des Multiway Stream Joins präsentiert, welche die vorgestellten Pipelines in eine parallele unidirektionale flow-basierte Architektur integriert (Parallel-MJ).

Als weiteren wichtigen Operator für den Queryprozessor, wird eine neuartige Hash-basierte Hardwarelösung zur Beschleunigung von Equi-Joins auf Streams (HB-SJ) vorgestellt. HB-SJ verwendet dazu zwei Murmur3 Hashfunktionen mit vier Speichertabellen. Falls die ausgewählte Reihe in allen vier Tabellen gefüllt ist, wird der neue Datensatz in einen Überflusspuffer eingefügt. Um die Ordnung der Datensätze nach Ankunft zu garantieren wird ein Sliding Window eingesetzt. Durch eine angepasste Konfiguration für HB-SJ werden extrem schnelle Sliding Window Zugriffe unabhängig der Größe erreicht.

Hardwarebeschleuniger bieten gute Leistung bei niedrigem Energieverbrauch. Hohe Forschungs- und Entwicklungskosten sowie lange Zeiten bis zur Marktreife, die sich für jede neue Anwendung wiederholen, beeinträchtigen jedoch aktuell ihren Erfolg. Im letzten Teil dieser Dissertation wird Simplex Stream Processor (SSP), der Nachfolger des nutzerspezifischen Queryprozessors, vorgestellt. Dieser basiert auf modularen Komponenten um die Echtzeit Verarbeitung von Datenströmen zu erreichen. Diese basiert wiederum auf dem zuvor beschriebenen unidirektionalen Datenflussmodell. SSP definiert Bibliotheken für das Kommunikationsnetzwerk und die Verarbeitungsblöcke mit einer definierten Schnittstelle, welche das nachträgliche Hinzufügen von Komponenten ermöglicht. Als Machbarkeitsnachweis haben wurde eine modifizierte Version der dritten TPC-H Query mit SSP relisiert und gemessen.

Acknowledgments

The work towards this doctoral dissertation took place at the Department of Informatics of the Technical University of Munich under the supervision of Prof. Dr. Hans-Arno Jacobsen. I would like to express my sincere gratitude to my supervisor Prof. Hans-Arno Jacobsen for the continuous support and guidance during my research, and for entrusting me as receiver of part of the hard earned funding from his Alexander von Humboldt Professorship. He gave me a great deal of freedom in terms of the direction of my research and also exercised pressure at those key moments when larger achievements were possible.

Contents

Abstract iii

Zusammenfassung v

Acknowledgments vii

1 Introduction 1
- 1.1 Motivation 4
- 1.2 Problem Statement 5
- 1.3 Approach 7
  - 1.3.1 Configurable Stream Processor 8
  - 1.3.2 Parallel Stream Join Architecture 8
  - 1.3.3 Circular Pipeline Design for Multiway Stream Joins 10
  - 1.3.4 Simplex Stream Processor on a Custom Network-on-Chip 12
- 1.4 Contributions 13
- 1.5 Organization 16

2 Background 17
- 2.1 Stream Processing Model 17
- 2.2 Stream Join Operation 18
- 2.3 FPGA Hardware Platform 19

3 Related Work 21
- 3.1 Parallel Stream Join 21
- 3.2 Multiway Stream Join 22
- 3.3 Hardware Stream Processor 24

4 Configurable Stream Processor 29
- 4.1 Stream Processor Design 30
  - 4.1.1 OP-Block Design Overview 30
  - 4.1.2 Instruction Format 31
  - 4.1.3 Architecture 33

4.1.4 Operation Example . . . 35
4.1.5 Design Rationale . . . 36
4.2 FQP Configuration Examples . . . 39
4.3 Segment-at-a-time Processing . . . 42
4.4 Variable Tuple Size Example . . . 44
4.5 Experimental Results . . . 45
4.5.1 OP-Block Analysis . . . 47
4.5.2 Query Insertion and Modification Latencies . . . 47
4.5.3 Performance Evaluation . . . 49
4.5.4 Scalability Evaluation . . . 49
4.5.5 Power Consideration . . . 51
4.5.6 Synthesis Challenges . . . 51
4.5.7 Segment-at-a-time Processing . . . 52

5 Parallel Stream Join Architecture **55**
5.1 SplitJoin . . . 56
5.1.1 Overview . . . 56
5.1.2 SplitJoin Parallelism . . . 59
5.1.3 Scalable Distribution Tree . . . 60
5.1.4 Expiration and Replacement Policies . . . 60
5.1.5 SplitJoin Algorithms . . . 61
5.2 Punctuated Result Collection . . . 62
5.2.1 Punctuation-based Ordering . . . 63
5.2.2 Ordering Algorithm . . . 65
5.3 Runtime Complexity . . . 66
5.3.1 Low-latency Handshake Join Analysis . . . 66
5.3.2 SplitJoin Analysis . . . 67
5.4 SplitJoin in Hardware . . . 69
5.5 Flow-based Stream Join in Hardware . . . 70
5.6 Experimental Results . . . 75
5.6.1 Experimental Setup . . . 75
5.6.2 Performance and Scalability . . . 76
5.6.3 Latency Evaluations . . . 78
5.6.4 Count-based Sliding Window . . . 80
5.6.5 Effect of Selectivity . . . 81
5.6.6 Effect of Punctuation Precision . . . 81
5.6.7 Hardware Splitjoin Evaluations . . . 82

6 Scalable Multiway Stream Joins in Hardware **87**
6.1 Multiway Stream Join . . . 88
6.1.1 Multiway Join with a Circular Pipeline . . . 89
6.1.2 Circular Pipeline Design Rationale . . . 90

6.1.3 Circular-MJ Architecture . . . 91
6.1.4 Circular-MJ Operation . . . 93
6.2 Multiway Stream Join with Stash . . . 95
6.2.1 Stashed-MJ Design and Rationale . . . 96
6.2.2 Stash Internal Architecture and Operation . . . 99
6.2.3 Stashed-MJ Analysis . . . 100
6.2.4 Parallel Multiway Stream Join . . . 102
6.3 Stashed-MJ Extensions & Discussions . . . 104
6.3.1 Stashed-MJ Window Access Pattern . . . 104
6.3.2 Stash Storage Pattern in the Invalid State . . . 104
6.3.3 Stash with Recomputation Support . . . 105
6.4 Supplemental Material . . . 108
6.4.1 Multiway Stream Join Architectural Challenges . . . 108
6.4.2 Effects of Parallelization on Match Burst Rate . . . 109
6.4.3 Time-Based Multiway Stream Join . . . 109
6.4.4 Extending the Buffering (Stash) for All Streams . . . 109
6.4.5 Hardware Optimization Using a Stash . . . 110
6.4.6 Extending Stash for Parallel-MJ . . . 110
6.4.7 Stage Buffer Potentials in Circular-MJ . . . 111
6.4.8 Ensuring Match Result Correctness Using the Control Unit . . . 111
6.4.9 Rationale to Fix the Join Order as the Right-Deep Join Tree . . . 112
6.5 Experimental Results . . . 113
6.5.1 Circular-MJ Evaluations . . . 113
6.5.2 Stashed-MJ Evaluations . . . 119
6.5.3 Parallel-MJ Evaluations . . . 122

7 Simplex Stream Processor on a Custom Network-on-Chip **125**
7.1 SSP Architecture . . . 126
7.1.1 System Vision . . . 126
7.1.2 Topology Brick . . . 127
7.1.3 Data Handling and Processing Blocks . . . 127
7.1.4 OP Block . . . 127
7.1.5 Complementary Custom Blocks . . . 129
7.1.6 Synchronizer Blocks . . . 132
7.1.7 Parallel Processing . . . 133
7.2 SCNoC Architecture . . . 133
7.2.1 Background . . . 134
7.2.2 Main Drawback . . . 135
7.2.3 Design Rationale . . . 136
7.2.4 GSwitch . . . 136
7.2.5 LSwitch . . . 137
7.2.6 Collector . . . 138

7.2.7 Instruction Set 139
7.2.8 Network Interface 139
7.3 Experimental Results 141
7.3.1 TPC-H Third Query Prototype 141
7.3.2 Throughput Measurements 145
7.3.3 Evaluating the Implementation 149
7.3.4 Power Consumption Evaluations 150
7.3.5 Implementing Stateless Queries 150

8 Conclusions **151**
8.1 Summary 152
8.2 Future Work 153

List of Figures **155**

List of Tables **159**

Bibliography **163**

Chapter 1

Introduction

Traditionally, the data management community has developed specialized solutions that focus on either extreme of a data spectrum between volume and velocity, yielding to large-volume batch processing systems, e.g., Hadoop [1], Spark [104], and high-velocity stream processing systems, e.g., Flink [20], Storm [87], Spark Streaming [105], respectively. However, the current trend, led by the Internet of Things (IoT) paradigm, leans towards the large-volume processing of rich data produced by distributed sensors in real-time at a high velocity for reactive and automated decision making. While the aforementioned specialized platforms have proven to achieve a certain balance between speed and scale, their performance is still inadequate in light of the emerging real-time applications.

This remarkable shift towards *big data* presents an interesting opportunity to study the interplay of software and hardware in order to understand the limitations of the current co-design space for distributed systems, which must be fully explored before resorting to specialized systems such as ASICs, FPGAs, and GPUs. Each hardware accelerator has a unique performance profile with enormous potential for speed and size to compete with or to complement CPUs, as envisioned in Figure 1.0.1. In this work, we primarily focus on cost-effective, power-efficient hardware acceleration solutions which excel at analytical computations by tapping into inherent low-level hardware parallelism. To motivate the adoption of these emerging hardware accelerators, we describe the four primary challenges faced by today's general-purpose processors.

Large & Complex Control Units —

The design of general-purpose processors is based on the execution of consecutive operations on data residing in main memory. This architecture must guarantee a correct sequential order execution. As a result, processors include complex logic to increase performance (e.g., super pipelining and out-of-order execution). Any performance gain comes at the cost of devoting up to 95% of resources

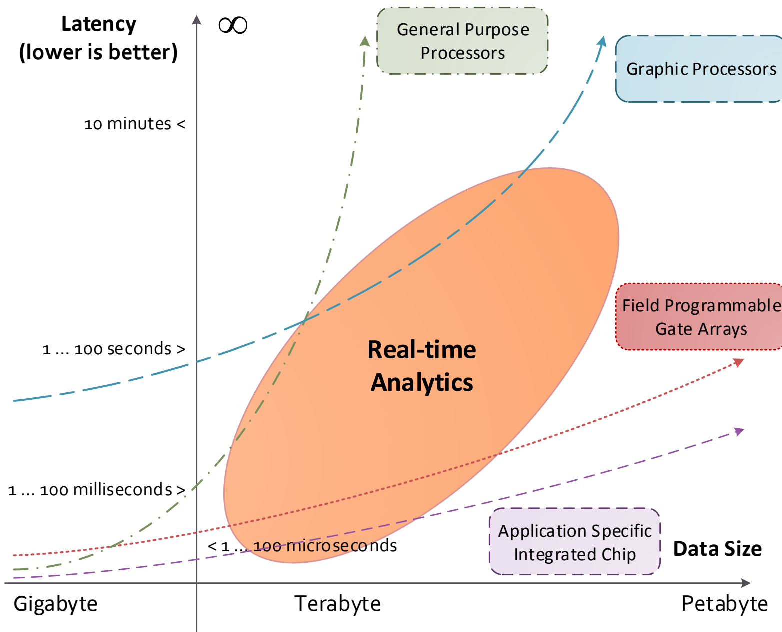


Figure 1.0.1: Envisioned acceleration technology outlook.

(i.e., transistors) to these control units [103].

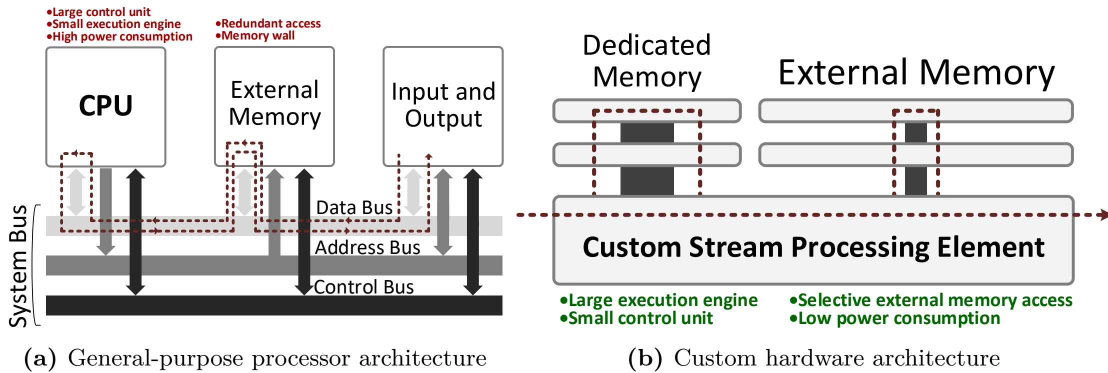
Memory Wall and Von Neumann Bottleneck —

The current computer architecture suffers from the limited bandwidth between CPU and memory. This issue is referred to as the memory wall and is becoming a major scalability limitation as the gap between CPU and memory speed increases. To mitigate this issue, processors have been equipped with large cache units, but their effectiveness heavily depends on the memory access patterns. Additionally, the von Neumann bottleneck further contributes to the memory wall by sharing the limited memory bandwidth between both instructions and data.

Redundant Memory Accesses —

The present-day system enforces that data arriving from an I/O device is first read/written to main memory before it is processed by the CPU, resulting in a substantial loss of memory bandwidth. Consider a simple data stream filtering operation that does not require the incoming data stream to be first written to main memory. In theory, the data arriving from the I/O should be streamed directly through the processor; however, today's computer architecture prevents this basic modus operandi.

Power Consumption —



(a) General-purpose processor architecture **(b)** Custom hardware architecture

Figure 1.0.2: General-purpose processor architecture vs. custom hardware architecture.

Manufacturers often aim at increasing the transistor density together with higher clock speed, but the increase in the clock speed (leading to a higher deriving voltage for the chip) results in a superlinear increase in power consumption and a greater need for heat dissipation [4].

These performance limiting factors in today's computer architecture have generated a growing interest in accelerating distributed data management and data stream processing using custom hardware solutions [86, 63, 44, 48, 76, 66, 101, 67, 98] and more general hardware acceleration at cloud-scale [79, 19, 41, 96, 72, 21]. In our past work on designing custom hardware accelerators for data streams (see Figure 1.0.2), we have demonstrated the use of a much simpler control logic with better usage of chip area, thereby achieving higher performance per transistor ratio [66, 67]. We can substantially reduce the memory wall overhead by coupling processor and local memory, instantiating as many of these components as necessary. Through this coupling, the redundant memory access is reduced by avoiding to copy and read memory whenever possible. The use of many low-frequency, but specialized, chips may circumvent the need for generalized but high-frequency processors. Despite the great potentials of hardware accelerators, there is a pressing need to navigate this complex and growing landscape before solutions can be deployed in a large-scale environment.

However, today's hardware accelerators comes in variety of forms as demonstrated in Figure 1.0.3, ranging from embedded hardware features, e.g., hardware threading and Single-Instruction Multiple Data (SIMD), in general-purpose processors (e.g., CPUs) to more specialized processors such as GPUs, FPGAs, ASICs. For example, graphics processing units (GPUs) offer massive parallelism and floating-point computation based on an architecture equipped with thousands of lightweight cores coupled with substantially higher memory bandwidth compared to CPUs. Due to the fixed architecture of GPUs, only specific types of applications or algorithms can benefit from its superior parallelism, such as tasks that follow a regular pattern (e.g., matrix computation). In contrast, a field-programmable gate array (FPGA) is an integrated circuit that can be configured to encode any algorithm, even irregular ones. FPGAs can be configured to construct an optimal architecture of custom cores engineered for a targeted algorithm. FPGAs contain compute components which are configurable logic blocks (CLBs) consisting of Lookup Tables (LUTs) and SRAM; memory

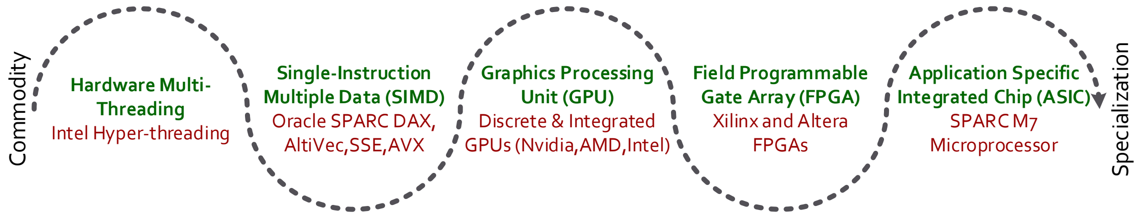


Figure 1.0.3: Accelerator spectrum.

components including registers, distributed RAMs, and Block RAMs (BRAMs); and communication components that consist of configurable interconnects and wiring. Any circuit design developed for FPGAs can be directly burned onto application-specific integrated circuits (ASICs), which provide greater performance in exchange for flexibility. Although the reminder of this work focus on FPGAs, many parts of the content discussed are equally applicable to other forms of acceleration.

This work targets the hardware acceleration of data processing with a focus on data streams by providing a novel custom processor architecture that provides two forms of flexibilities. The first form comes from having modular, processing and communication, components that are rearrangeable to construct a custom topology for a specific application. The second form is realized by providing the re-programmability feature in the communication network and some of processing components to make it possible to adapt a running instance of our solution to new changes and demands in real-time.

The goal of our solution is to drastically reduce cost, complexity, and time-to-market factors while presenting extreme benefits of hardware acceleration, i.e., superior performance and power consumption.

1.1 Motivation

Digital technology, as well as its challenges and opportunities, is growing at a rapid pace [50, 52, 83, 6, 9]. This growth is not only in the number of connected devices but also in the velocity and volume of the collected data.

On the positive side, a large part of information in the digital world is unimportant and transient [89] (e.g., unsaved phonecalls). Furthermore, in practice, we overlook less important data because of limitations on storage and processing technology. This situation has spawned real-time stream analysis and similar concepts.

As an example of these limitations, we refer to the square kilometre array (SKA), a large radio telescope project planned for construction in Australia, New Zealand, and South Africa. It will

have a total collection area of approximately one square kilometer. The SKA will collect in excess of an exabyte of data per day [55]. Research is already underway on how to handle such a large volume of data, such as approaches to the real-time detection of binary pulsars [93]. A common feature of most such research is that it explores the processing of transient data to extract more valuable data. This can be in the form of evaluation results or refined raw data that, owing to the reduced volume, can be stored and trivially reprocessed later.

The storage and processing limitations as well as the remarkable shift toward *big data* present an interesting opportunity to study the interplay of software and hardware. This can help us understand the advantages and disadvantages of the current hardware and software co-design space for distributed systems, which must be fully explored before resorting to specialized systems such as application-specific integrated circuits (ASIC), field-programmable gate arrays (FPGAs), and graphics processing units (GPUs). Each hardware accelerator has a unique performance profile with enormous potential to compete with software solutions or complement them [24].

The adoption of hardware accelerators, particularly FPGAs, is gaining momentum in both industry and academia. Such cloud providers as Amazon are building new distributed infrastructures that offer FPGAs[1] connected to general-purpose processors (CPUs) using a PCIe connection [71]. Furthermore, the FPGAs employed share the same memory address space as the CPUs, which can spawn new applications of FPGAs using co-placement and co-processor strategies. Microsoft's configurable cloud [21] also uses a layer of FPGAs between network switches and servers to filter and manipulate dataflows at line rate. Another prominent deployment example is IBM's FPGA-based acceleration within the SuperVessel OpenPOWER development cloud [5]. Google's tensor processing unit (TPU), designed for distributed machine learning workloads, is also gaining traction in its data centers, although current TPUs are based on ASIC chips [8].

1.2 Problem Statement

Despite enormous throughput and latency benefits, custom hardware also comes with its own drawbacks. The first drawback is the difficulty of adapting the hardware to changes in currently utilized algorithms or adopting new approaches. If these changes are not foreseen in the initial design, it will be a complex and time consuming task to redesign the system to apply/include the changes, assuming that we are dealing with re-programmable hardware such as FPGAs and not application specific integrated systems (ASICs).

The second drawback is the difficulty of upgrading the hardware platform, as the custom design utilized also needs to be readjusted and redesigned for the new platform. This issue arises since there

[1] Xilinx UltraScale+ VU9P fabricated using a 16 nm process, and with approximately 2.5 million logic elements and 6,800 digital signal processing (DSP) engines.

is no default/base architecture to handle the compatibility between various hardware generations. Although defining some standards in the hardware design makes it easier to upgrade the platform later, this poses a risk to the efficiency of newer designs and also to the complexity of hardware design and development. Additionally, there is a large amount of time required to redesign and rebuild hardware to accommodate a new approach. This means that if a solutions is needed quickly and is not expected to be used for a long period, it may not be a good choice to go for a hardware solution.

The third drawback is the complexity of a hardware solution integration into the rest of the system. To mitigate this drawback, the common practice is to bundle the hardware with general purpose processors using standard interfaces such as PCIe. Although using FPGAs in the computer architecture, built with generality in mind, degrades some of their properties such as power consumption (considering the whole system), still, the boosted performance and the access to additional features provided by general purpose systems makes this practice favorable.

In this dissertation, we focus on distributed real-time analytics over continuous data streams using FPGAs. To cope with the lack of flexibility of custom hardware solutions, state-of-the-art approaches assume that the set of queries is known in advance. Essentially, past works rely on the compilation of static queries and fixed stream schemas onto hardware designs synthesized to configure FPGAs [63]. These hardware-accelerated approaches are characterized by a processing pipeline that synthesizes static (sets of) queries into circuits operating on a FPGA. Since some of the inner operations (i.e., logic-optimization and technology mapping) in the synthesis process are NP-hard [29], existing synthesis algorithms are heuristic in nature. They commonly suffer from the drastic increase in synthesis time as designs grow in size and complexity, and the synthesis often takes hours to days to complete. As a result, most FPGA-based solutions are too inflexible to support ad-hoc and interactive queries that must be able to change operations on the fly.

Designing hardware based on the compilation of static queries is not suitable for modern-day stream processing needs, which require fast, on the fly reconfiguration. Furthermore, many of the existing approaches assume a complete system halt during any synthesis modification [63, 76]. While synthesis and stream processing may overlap to some extent, a significant amount of time and effort is still required to reconfigure, which may take from several minutes up to hours. More importantly, this approach requires additional logic for buffering, handling of dropped tuples, requests for re-transmissions, and additional data flow control tasks, all of which renders this style of processing difficult, if not impossible, in practice. These concerns are often ignored in the approaches listed above, which assume that processing stops entirely before a new query-stream processing cycle starts.

Even if the latency hit due to synthesis could be tolerated in certain domains, substantial time and effort is still required to halt, re-configure, and resume operation. Also, extra logic is required for buffering, handling of dropped tuples, requests for re-transmissions, and other non-trivial data flow

control logic to resume the operation. Therefore, any practical hardware solution must deliver a flexible design with acceptable processing performance.

1.3 Approach

In this work, we fill the gap between software solutions that are known to provide the greatest degree of flexibility and hardware solutions that are famous for their massive performance and efficiency.

As the first step towards our goal, we propose a flexible query processor, namely FQP, that uses rearrangeable components for a straightforward customization. As one of the key building blocks, FQP benefits from an online programmable-block (OP-Block) which adds support for real-time updates in the execution instructions. The FQP data-path is specially designed to support stream join parallelization as one of the most resource-intensive operators in stream processing. Unfortunately, this limits the flexibility of our processor by forcing a bidirectional data-flow design for its data-path. From these limitations, we can refer to the challenging operation of tasks (to processing blocks) assignment. As a result of the bidirectional design of the data-path, there are less available placement choices which increase this operation complexity.

As the second step, we address the problem imposed by the bidirectional data-flow by proposing SplitJoin a scalable architecture for stream join parallelization based on a unidirectional data-flow. SplitJoin also provides a scalable ordering technique for the results, which is crucial for some real-time applications.

Since most queries deal with multiway joins, as our third step we propose a novel scalable architecture, namely Circular-MJ, to add support for this essential operator in our query processor. Additionally, we design and develop a novel hash-based equi-join operator (HB-SJ) to additionally support ultra-fast processing using an indexing technique.

As the final step in this work, we propose a novel redesigned successor to our FQP based on the unidirectional data-flow, referred to as simplex stream processor (SSP). This processor is constructed on top of a network-on-chip, customized for streams, that allows addition or modification of processing components in a straightforward way.

In the following, we describe the aforementioned steps in more details.

1.3.1 Configurable Stream Processor

We propose a streaming architecture composed of a configurable number of stream processing elements called online programmable-blocks (OP-Blocks) that accept new queries in an online fashion without disrupting the processing of incoming data streams. Essentially, input queries are interspersed into the data input stream and update the processing on-the-fly. While supporting query modifications at run-time is trivial for software-based techniques, they are not well-studied for hardware-based approaches. Moreover, our design allows for OP-Blocks to be statically inter-connected to form different topologies, specifically tailored to the queries being processed.

Together with a number of auxiliary components for query and tuple buffering, routing, and dispatching, the OP-Blocks form an instance of the FQP that operates entirely on the FPGA. The inter-connection topology for the OP-Blocks can be chosen in the manner most advantageous for the queries to be processed. For example, if the query workload lends itself for parallelism, a parallel topology can be chosen, whereas for workloads with more data dependency, a pipelined topology can be chosen. The choice of topology is performed statically and an instance of FQP is synthesized that realizes this topology. The OP-Block is the processing core that implements the actual query operators. It enables online changes to queries based on a number of parameters, including variable tuple size, projection attributes, selection conditions, join conditions, and join-window size.

In the design of FQP, we dealt with a number of challenges: First, a static FPGA-based query processor must over-provision resources to handle the largest expected (intermediate) tuple size, which under-utilizes system resources. Second, the change in tuple size between the join operation's inputs and output adds new challenges, especially when there is the need to use the join result as input for other operations. Third, determining a minimal processing core that can efficiently handle a variety of query operators, some of which are stateless, while others are stateful.

1.3.2 Parallel Stream Join Architecture

Scalable stream processing is an integral part of a growing number of data management applications such as real-time data analytics [82], algorithmic trading [77], intrusion detection [31], and targeted advertising [35]. What is common among these scenarios is a predefined set of streaming queries (e.g., ad campaigns or trading strategies) and an unbounded event stream of incoming data (e.g., user profiles or stock feeds) that must be processed against the queries in real-time.

In data stream processing, the execution of continuous queries (i.e., repetitive tasks) on potentially unbounded streams using a finite-window semantics provides a key motivation for acceleration. In the context of stream processing stream join parallelization has received great attention due to its computational complexity [39, 85, 27, 45, 90]; still, the existing hardware approaches suffer from

lack of scalability for practical use.

Besides leveraging hardware acceleration, coping with the high-velocity of unbounded incoming streams has forced the stream operation model to shift away from the traditional “store and process” model that has been prevalent in database systems for decades. However, the mindset of sequential stream join processing (or constructing lengthy processing pipelines) and, essentially, thinking of a stream as a sliding window (or a long chain of sequentially incoming tuples to resemble database relations) has continued to shape the way stream processing is carried out today, even on low-latency and high-throughput stream processing platforms.

We tackle two main shortcomings of existing stream join processing architectures: the sequential operation model (i.e., “store” and “process”) and the linear data flow model (i.e., “left-to-right” and “right-to-left” flows). We propose SplitJoin, the first step in re-thinking the stream join operation model, which is built on the implicit assumption that storage of newly incoming data, whether stored in a relation or a memory buffer, must always precede processing. Instead, we abstract the computation steps as two independent and concurrent steps, namely, (i) “storage” and (ii) “processing”.[2] This new splitting abstraction of join cores enables unprecedented scalability by allowing the system to distribute the execution across many independent storage cores[3] and processing cores. Second, we change the way tuples enter and leave the sliding windows, namely, by dropping the need to have separate left and right data flows (bidirectional flow). SplitJoin introduces a novel top-down data flow (unidirectional flow), where incoming tuples (from both streams) are simply arriving via the same path downstream (preserving input stream order), while the join results are further pushed and merged downstream using a novel relaxed adjustable punctuation (RAP) technique (preserving the output stream order). Unlike recent advances in stream join processing [39, 85, 76, 75], SplitJoin does not rely on central coordination for propagating and ordering the input/output streams.

SplitJoin’s top-down (unidirectional) data flow trivially satisfies the ordering of incoming tuples and eliminates the in-flight race condition between the left and right streams as tuples travel from one core to the next. Unlike existing approaches [85, 75], the top-down flow also eliminates the need for communication between the join cores. In SplitJoin, the top-down flow is realized using a distribution tree for routing incoming tuples into their corresponding sub-window that addresses the scaling issues of adding new join cores. The adopted distribution mechanism nicely fits into the coordination-free protocol of SplitJoin for distributing new tuples to both storage and processing cores. For example, all join cores receive the newly incoming tuples (achieving the desired expedited delivery, without the linear forwarding used in [75]), while only one storage core stores the new tuple. Both the storage and eviction of tuples to and from cores are done in a round-robin fashion; thus, naturally, in the same order that cores store a new tuple, they evict their oldest tuple (again, without any explicit coordination). This can be generalized to batches of tuples instead of a single

[2]In relational databases, tuples are first stored in relations prior to being processed (e.g., performing a join) while in a stream join, the incoming tuples are first processed and subsequently stored in sliding windows [53].

[3]A storage core is an abstraction for an in-memory sliding window, tightly coupled with a join core.

tuple as well.

SplitJoin has provably lower runtime complexity as compared to state-of-the-art parallel distributed join algorithms [85, 75]. SplitJoin exhibits an overall system latency of $O(\log_b k)$, where k is the number of join cores and b is the branching factor of the distribution tree. In contrast, the state-of-the-art handshake join has $O(k)$, while the original version resulted in an $O(n)$ latency, where n is the number of tuples a window can hold ($k \ll n$) [85, 75].

SplitJoin's coordination-free distribution also lends itself to a simpler resiliency against failures; for example, core failures do not halt or disrupt the entire join computation and affect only the failed nodes (the loss is limited to only failed nodes). In contrast, in a linear left-to-right data flow, if any cores fails, then, on average, half of the cores may not receive any data.

1.3.3 Circular Pipeline Design for Multiway Stream Joins

Considering the crucial role of joins as resource-intensive operators in relational databases, it is not of a surprise that stream joins have also been the focus of much research on data streams [53, 39, 85, 43, 28, 68]. For example, consider TPC-H [30] where 20 queries (out of 22) contain join operator while 12 of them use multi-way joins some up to 7 joins. However, the importance of joins is no longer limited to only the classical relational setting. The emergence of Internet of Things (IoT) has introduced a wide wave of applications that rely on sensing, gathering, and processing data from an increasingly large number of connected devices. These applications range from scientific and engineering domains to complex pattern matching methodologies [14, 73].

As an example, consider the deployment of smart meters to analyze households power consumption and weather forecast live feeds. By analyzing these live streams, we can evaluate the correlations between these two streams and how fast the weather forecast fluctuations affect the power consumption. Discovering these patterns are critical to plan and optimize power plants to improve their efficiency and reliability (e.g., avoid blackouts). In smart home setting, one can imagine enhancing the household with solar and wind power, each resulting in generating its own live feeds, that can be combined with the weather forecast and the power consumption feeds, thus, demanding the ability to join multiple data streams. As a result, coping with a multitude of data streams is unavoidable in the IoT when one expects to have millions to billions of connected devices [25, 32] to enable real-time analysis of their live feeds.

In stream processing, software platforms offer flexible communication, where we find anycast and multicast connections between internal components without a significant decrease in performance. As an example, consider a system with four internal components A, B, C, and D, where a point-to-point connection between them could form a data path similar to $A \rightarrow B \rightarrow C \rightarrow D$. In a software platform, establishing additional communications, i.e., $A \rightarrow D$ and $B \rightarrow D$, is not

detrimental, particularly given the flexible shared memory hierarchy. State-of-the-art software approaches in a multiway stream join benefit from this provided flexibility that leads to an unstructured[4] architecture. However, on hardware, it is essential to predetermine and plan the necessary communication channels; otherwise, design performance, complexity, and cost rapidly increase since all communications have to be exclusively realized using physical connections. This situation motivates our complete reconsideration of the hardware design rather than simply relying on the reimplementation of available software solutions.

From the hardware acceleration perspective, executing continuous queries (i.e., repetitive tasks) on potentially unbounded streams using finite-window semantics offers a unique opportunity. A hardware solution presents negligible, if any, gains when executing an operation only once versus its competing software variant; however, when this operation repeats many times, the amortized gains increase far beyond that of a software variant.

It is nontrivial to build multiway join operators by cascading operators designed for two streams because each new tuple, depending on its origin, requires its own order of join operators for processing. In a straightforward hardware design, this requirement leads to an unstructured design due to the required anycast connections between the join operators. Therefore, designing a scalable (with respect to the number streams) architecture for multiway stream joins in hardware remains a major challenge.

We propose a novel circular pipeline architecture for multiway stream joins (Circular-MJ), which is centered around direct neighbor-to-neighbor communication. We use two fundamental steps to reshape the problem of multiway joins to design a scalable hardware architecture. First, we transform the problem of unstructured multiway stream join design into a join reordering problem in a structured design. This means that we do not need anycast connections between the join operators. Second, we eliminate the join operator reordering problem by moving the reordering task to tuple insertion circuitry using a pipelined distribution chain. The operator reordering causes a scalability issue in hardware that is eliminated by this step.

Depending on the streams' characteristics and the join operators' properties, materializing intermediate results[5] in a buffer may provide performance improvements by avoiding the recomputation of already processed tuples. In our proposed Circular-MJ, the processing engine for each stream is placed in a separate stage, and accessing this buffer from independent pipeline stages imposes a resource sharing challenge in a hardware realization. First, having a shared unit between multiple stages violates the main concept of pipeline design, which is the separation of concerns. Second, this sharing can result in race conditions between the storage and processing of multiple tuples at different stages, which then require expensive stalls in the pipeline.

[4] We refer to an architecture as unstructured when internal nodes have anycast, multicast, or broadcast connections within the system.

[5] Outputs of each join operator inside a join tree.

To avoid the recomputation, we propose Stashed-MJ, a custom two-stage pipeline that includes a *stash*[6]. The novelty of our approach is to benefit from the reduction in the number of pipeline stages (two rather than three) in favor of better utilizing the available processing units and avoiding the recomputation of already processed data. Note that there is always one idle processing unit for the stage that hosts the current tuple sliding window in the three-stage design. Therefore, the processing unit in the first stage operates on two windows but not simultaneously, which also eliminates the resource sharing challenge.

Additionally, to scale up the processing throughput of our multiway stream join, we propose Parallel-MJ, an enhanced design of our multiway join pipeline mapped into a unidirectional flow-based parallelized architecture suggested by our SplitJoin.

1.3.4 Simplex Stream Processor on a Custom Network-on-Chip

Many solutions in stream processing have been reported in recent years, such as [17, 15, 62, 23, 31, 47, 33], which are primarily software approaches, and [63, 78, 36, 12, 101], which also benefit from hardware platforms. However, many of these approaches provide either no support or only limited support for hardware acceleration. The main reason for this limitation is the complex and time-consuming process of hardware design, development, and adoption, which even renders the hardware solutions inapplicable because some applications and demands tend to update or change frequently.

This part of the dissertation focuses on reducing the complexity of the hardware design in stream processing systems, and the time gap between new demands and the adoption of a hardware approach. We propose a simplex stream processor (SSP) that benefits from a stream customized network-on-chip (SCNoC) and a library of hardware components as well as a predefined interface and methodology to re-customize the available components or add new ones.

There are three key differentiating factors that separate SSP from its predecessor, flexible query processor (FQP) [66, 67].

The FQP uses a bidirectional dataflow (Figure 7.1.2a) that is necessary for the join parallelization capability of the stream. This requires challenging mappings to place operators and route intermediate results from the output of an operator to the input of another one.

The SSP architecture uses a unidirectional dataflow (Figure 7.1.2b) that is not dedicated to any specific type of stream. This architecture is made possible by the introduction of join parallelization to the stream using a single data path in [68]. As a result of this architecture, the processing components can be simply placed one after another as opposed to the arrangement in the FQP.

[6] We refer to an intermediate result buffer with additional control circuitries as a stash.

In the SSP architecture, the concepts of the distribution network and processing components are realized independently. We subsequently use modular interfaces to connect the distribution network and processing components. This type of implementation provides the simplicity of adding processing components where needed in the distribution network, which routes corresponding streams to each component using defined instruction sets. This allows for the straightforward (no need for complex mappings) selection of an architecture for a set of queries. By contrast, in the FQP, queries are placed on a rigid chain of processing components using a complex mapping algorithm. Moreover, a set of queries targeting a specific area of processing may not even require many stream join operators. However, the FQP architecture was built on two data paths (Figure 7.1.2a), and therefore remains mainly underutilized because one (mandated by the join operation) of the data paths remains idle. In SSP the bandwidth of all available data paths is well utilized because it is not limited to specific operations.

The SSP architecture is designed based on two (one input and one output) port components for processing and transmission, whereas the FQP has components with five (two input and three output) ports. In terms of hardware solutions, the number of ports has a close relation to the cost, complexity, and efficiency of a component. In SSP, each component implements a consume-and-produce model that simplifies the system architecture. This change has removed the need for large and complex switching and control circuitries responsible for consistently receiving, processing, and transmitting streams and results from multiple ports.

In SSP, tuples for distinct streams involved in the join operation arrive in order, whereas in the FQP, there are two separate paths, and this leads to out-of-order arrivals. Consequently, complex buffering and controlling circuitries are required to ensure the consistency of the results.

The aforementioned factors render SSP a practical framework for constructing a query processor with various degrees of flexibility that allow for fine-grained and coarse-grained adjustments for specific applications.

1.4 Contributions

We present the following contributions from our flexible query processor (FQP):

i. We design a parameterizable stream processing element, called OP-Block, that lies at the core of a hardware-based flexible query processor (FQP). Through hardware specialization and dividing the chip area into many small, self-contained, independent OP-Block instantiations, we eliminate the need for large and overly complex control units for coordination, and achieve better chip area utilization.

ii. Enable the fine-grained dynamic reconfiguration of the OP-Block that allows the application to re-program it in an online fashion without disrupting query processing.

iii. Present the coarse-grained static reconfiguration of the OP-Block inter-connection topology that allows us to attach and connect several OP-Blocks to realize complex queries and to adapt to the specific processing needs of these queries.

The main contributions from our stream join parallelization work, SplitJoin, are:

i. We propose a scalable architecture for stream join parallelization, called SplitJoin, that removes inter-core communications and dependencies. SplitJoin introduces a unidirectional data-flow model to achieve a coordination-free protocol which eliminates the need for the limiting bidirectional data-flow model used in the original FQP.

ii. Present a coordination-free protocol that does not rely on global knowledge to produce ordered join output streams by proposing a relaxed adjustable punctuation technique with tunable precision.

iii. Provide the hardware (in addition to the software) design and development of SplitJoin, while evaluating its advantages in regard to the previous approaches.

In our multiway stream joins work we make the following contributions:

i. We propose a scalable multiway stream join (Circular-MJ) on hardware that is built on a circular chain of dedicated stages (one per stream) and that benefits from pipeline parallelism.

ii. We present a novel two-stage pipeline (Stashed-MJ) that benefits from a stash (intermediate results buffer) to accelerate processing throughput.

iii. We enhance our multiway stream join pipeline to integrate it into the parallel architecture to linearly scale the processing capabilities of our solution (Parallel-MJ).

Finally, for our simplex stream processor, referred to as SSP, we make the following contributions:

i. We propose a stream customized network-on-chip (SCNoC) with instruction sets for routing instructions and tuples.

ii. We propose SSP with a configurable topology (placement of processing components in the distribution network) and instruction sets built on our SCNoC.

iii. We propose a hash-based stream join (HB-SJ) architecture for fast equi-join processing and integrate it into a modified version of our circular multiway stream join (Circular-MJ) architecture.

iv. We present a hardware architecture for the aggregation and groupby operators.

v. We present a mapping of and implement the TPC-H on a new instance of SSP and evaluate it based on certain performance metrics.

Parts of the content and contributions of this work have been published in or have been submitted to the following venues:

- M. Najafi, M. Sadoghi and H.-A. Jacobsen. Flexible query processor on FPGAs. Journal Proceedings of the VLDB Endowment, Volume 6 Issue 12, pages 1310-1313, 2013
- M. Najafi, M. Sadoghi and H.-A. Jacobsen. Configurable hardware-based streaming architecture using Online Programmable-Blocks. 31st IEEE International Conference on Data Engineering (ICDE), pages 819-830, 2015
- M. Najafi, M. Sadoghi and H.-A. Jacobsen. The FQP Vision: Flexible Query Processing on a Reconfigurable Computing Fabric. ACM SIGMOD Record, Volume 44 Issue 2, pages 5-10, 2015
- M. Najafi, M. Sadoghi and H.-A. Jacobsen. SplitJoin: A Scalable, Low-latency Stream Join Architecture with Adjustable Ordering Precision. USENIX Annual Technical Conference (USENIX ATC 16), pages 493-505, 2016
- M. Najafi, K. Zhang, M. Sadoghi and H.-A. Jacobsen. Hardware Acceleration Landscape for Distributed Real-Time Analytics: Virtues and Limitations. 37th IEEE International Conference on Distributed Computing Systems (ICDCS), pages 1938-1948, 2017
- M. Najafi, M. Sadoghi and H.-A. Jacobsen. A Scalable Circular Pipeline Design for Multi-Way Stream Joins in Hardware. 34th IEEE International Conference on Data Engineering (ICDE), pages 1280-1283, 2018
- M. Najafi, M. Sadoghi and H.-A. Jacobsen. Scalable Multiway Stream Joins in Hardware. IEEE Transactions on Knowledge and Data Engineering (TKDE), 2019 (accepted for publication)
- M. Najafi and H.-A. Jacobsen. A Simplex Stream Processor on a Custom Network-on-Chip. Submitted to IEEE Transactions on Knowledge and Data Engineering (TKDE), 2019

1.5 Organization

The rest of this document is organized as follows. Chapter 2 presents related work in the area of hardware acceleration with emphasis on stream processing. Chapter 3 provides background on field programmable gate arrays (FPGAs) as the common hardware acceleration framework. Chapter 4 explores the design and realization of our flexible query processor, referred to as FQP. Chapter 5 starts with the design and realization of our parallel stream join in software and ends with its scalable hardware design, realization, and comparisons. Chapter 6 presents our novel circular multiway stream joins hardware architecture, referred to as Circular-MJ, and continues by proposing an optimized architecture which avoids recomputing the processed data using a best-effort buffering technique, referred to as Stashed-MJ. Finally, Chapter 7 presents our simplex stream processor, referred to as SSP, and includes an unidirectional custom communication network while presenting the design and realization of our other major components such as: a hash-based stream join, an orderby operator, and an aggregation-groupby operator.

CHAPTER 2

Background

2.1 Stream Processing Model

Our data stream processing model follows an attribute-value pair form, which closely resembles a database tuple. Our data stream language follows the traditional database SPJ queries including selection (σ), projection (π), and join ($\bowtie$, $\bowtie_\theta$) operators.

In fact, we use a subset of the PADRES SQL (PSQL) [34], an expressive SQL-based declarative language for registering continuous queries against data streams over a count-based sliding window model. Essentially, the sliding window is a snapshot of an observed finite portion of the data stream. Formally, the stream processing model is defined as follows: *Given a stream of tuples and a set of SPJ queries, execute the queries continuously over the data stream and output the resulting tuples.*

Join operator: For join, there are two input streams, R and S. Each time a new tuple arrives in the window of one stream, the tuple is compared against all existing tuples in the other stream's window. For each match, a result tuple consisting of the two matching tuples is produced.

Projection and selection operators: In contrast to the join operator, stateless operators, such as projection and selection, are executed on each tuple independently. For the selection operator, a tuple is not filtered, if it satisfies the selection condition, and for the projection, the resulting tuple includes only the projected fields.

The design presented in this work supports selection and projection operators over a single stream, join operators over two streams, and allows for arbitrary logic formula to express the selection and join conditions using $\{=, \neq, <, >, \leq, \geq\}$.

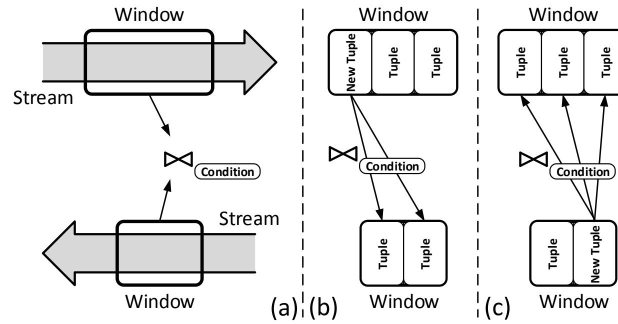


Figure 2.2.1: Sliding window concept in stream join.

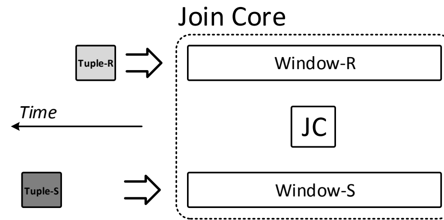


Figure 2.2.2: Traditional stream join architecture.

2.2 Stream Join Operation

The relational join (theta join) between two non-stream relations R and S, defined as $R \bowtie_\theta S$, produces the set of all resulting pairs (r, s), which satisfy the join condition $\theta(r, s)$ and $r \in R$, $s \in S$. Extending this definition to stream join implies the same join processing semantics with the exception that streams, unlike relations, are unbounded. To mitigate the challenge of unbounded streams, with respect to both processing and storage limitations, streams are conceptually seen as bounded sliding windows of tuples, as shown in Figure 2.2.1. The size of these windows are defined as a function of time or number of tuples, referred to as time-based or count-based windows, respectively.

Figure 2.2.2 shows the traditional architecture of a join operator that receives Tuple-R and Tuple-S from streams R and S, respectively. *JC* stands for join core, which performs the join operation. To process the tuples shown in the figure, Tuple-R is inserted into *Window-R*, then it is evaluated against all existing tuples in *Window-S* and the join results are returned. Similarly, Tuple-S is inserted into *Window-S* and the same join procedure is applied.

2.3 FPGA Hardware Platform

Field Programmable Gate Arrays (FPGAs) are reconfigurable chips that realize a high-level hardware design. Design entry starts with the modeling of the hardware design using a high-level hardware description language, such as Verilog, VHDL, or SystemVerilog. These allow users to specify the circuit behaviorally using traditional C/C++-style constructs for condition, loop, and data types.

Summary of FPGA Building Blocks —

FPGAs contain a matrix of interconnected discrete random access memory cells which can realize any Boolean function. Each memory cell, called a Look-Up-Table (LUT), is essentially implemented as a $2N$ bit array, where N is the number of address lines for the LUT. Each LUT is capable of implementing an N-input and 1-bit output Boolean function (circuit) by saving all possible output valuations for each N-input combination. Therefore, the LUT realizes a truth-table which is itself a fundamental component of the FPGA fabric that is inter-connected with other blocks to construct more practical and complex logical circuits.

An FPGA also offers two on-chip memory options: Block RAM (BRAM) and Distributed RAM (DRAM). BRAMs are discrete dual-ported single-cycle access latency, on-chip memories that offer several kilobits of storage. These can either be initialized with data at configuration time or be internally updated at run-time. Block RAM in the Xilinx Vertex 5 family of FPGAs is available as 36Kb x 1 discrete blocks but each can be used as two independent 18Kb x 1 blocks. Moreover, several BRAM blocks can be tied together to create wide-ported memories, without paying any fan-out penalty. DRAM is realized by configuring several LUT-based cells to act as memory. DRAM is expensive and is generally discouraged, as it reduces the logic real estate available on the FPGA and incurs higher signal routing cost.

FPGAs' internal memories offer low-latency access (on the order of the FPGA's operating clock frequency) and read/write from/to them does not translate to the so-called *redundant memory accesses* issue in a well distributed hardware design.

Design Synthesis Steps —

Once the design description is completed a synthesis tool is used to convert the circuit description into a netlist file which contains circuit implementation consisting of gate instances and connections (nets) between them. A netlist is platform independent and does not include any user specified timing constraints which are essential to ensure that a circuit meets the design requirements (e.g., operating frequency, data processing throughput etc.). Once the netlist has been generated, it needs to be customized for implementation on a specific target technology. This step is called, *design mapping* and is carried out by vendor specific tools that replace generic netlist gate instances with

appropriate instances from a hardware library that meet user specified timing, energy and device space constraints. The design mapped output is in vendor specific circuit format and is targeted to run on a particular FPGA technology.

Finally, a *place and route* tool maps the timing critical design components to the appropriate sections of the FPGA and optimizes on-chip signal routes. Both of these steps significantly impact the overall maximum clock frequency the design can be safely run at, without causing timing violations and other erratic behavior. In addition, with the knowledge of the underlying FPGA architecture, the initial design description can be varied to take advantage of the special features of the physical hardware and reduce signal routing cost to potentially speed up the design operation.

In the final step, the design is compiled and encoded into a bitstream file that is used to configure the FPGA. The bitstream can either be downloaded to the FPGA via the JTAG port or be stored on an on-board memory to configure the FPGA every time upon powerup. Lastly, it is important to note that all design steps are carried out in software through vendor-provided integrated development environments (IDE)[1].

[1] For example, the Integrated Software Environment (ISE), i.e., also known as IDE, is the Xilinx design software suite that offers a comprehensive front-to-back design environment from design entry through Xilinx device programming and verification, `http://www.xilinx.com/univ/dtools.htm`.

Chapter 3

Related Work

3.1 Parallel Stream Join

Work related to our approach can be broadly classified into stream join algorithms [39, 85, 75, 66, 53, 56], or more generally speaking, stream processing in software [23, 26, 7], approaches to performance-optimize stream processing through emerging hardware mechanisms [100], in particular, through FPGA-based acceleration [44, 63], but also, through GPUs and processor-based I/O processing innovations [88]. The survey [102] covers other related work and topics including concepts such as ordering in stream join. SplitJoin can be incorporated in any of the existing streaming engines (i.e., [13, 57, 2]).

Stream Join Algorithms —

An early stream join was formalized by Kang's three-step procedure [53]. Subsequently, Gedik et al. [39] introduced the parallel CellJoin, designed for a heterogeneous architecture, aiming to substantially improve stream join processing performance. However, CellJoin requires a re-partitioning task for each newly incoming tuple, which limits its scalability [39]. The problem of distributed stream join processing has also been studied with respect to elasticity and reduction of memory footprint, applicable to cloud computing [56].

Teubner et al. introduced a bi-direction data flow-oriented stream join processing approach, called the handshake join [85]. To reduce delay in the linear chaining, Teubner et al. [75] introduced a low-latency handshake join that uses a fast forwarding mechanism to expedite tuple delivery to all sub-windows by replicating every tuple k times, where the stream is split over k join cores. This mechanism is illustrated in Figure 5.3.1. Furthermore, the bi-directional flow complicates the

logic for serializing the two pipes connecting consecutive join cores that is necessary in order to avoid race conditions due to concurrent in-flight tuples (i.e., tuples traveling between neighboring processing cores).

Stream Processing Acceleration —

Stream processing has received much attention over the past few years. Many viable research prototypes and products have been developed, such as NiagaraCQ [26], TelegraphCQ [23], and Borealis [7], to just name a few. Most existing systems are fully software-based and support a rich query language, but stream join acceleration has not been the main focus of these approaches.

Since the inception of stream processing, the development of optimizations both at the query-level and at the engine-level have been widely explored. For example, co-processor-based solutions utilizing GPUs [88, 39] and more recently hardware-based solutions employing FPGAs have received attention [86, 44, 63, 99, 81]. For example, Tumeo et al. demonstrated how to use GPUs to accelerate regular expression-based stream processing language constructs [88]. The challenge in utilizing GPUs lies in transforming a given algorithm to use the highly parallel GPU architecture that has primarily been designed to perform high-throughput matrix computations and not, foremost, low latency processing.

Past work showed that FPGAs are a viable option for accelerating certain data management tasks in general and stream processing in particular [86, 44, 63, 99, 81, 18, 64]. For example, Hagiescu et al. [44] identify compute-intensive nodes in the query plan of a streaming computation. To increase performance in the hardware design that realizes the streaming computation, these nodes are replicated, which, due to the stateless nature of the query language considered, poses few issues. A main difference from our work is the restriction to stateless operations and the lack of a capability to flexibly update the streaming computation. Similarly, Mueller et al. [63] present Glacier, a component library and compiler, that compiles streaming queries into logic circuits on an operator-level basis. Both approaches are characterized by the goal of hardware-aware acceleration of streams, yet our solution is also applicable to non-FPGA parallel hardware.

3.2 Multiway Stream Join

Multiway join processing has received considerable attention because it is among a number of critical operations used in many applications, such as data mining, analytics and IoT [40, 60, 51, 46, 92, 10, 59, 110, 49, 42, 38, 109, 108, 106]. Among these approaches, the approaches proposed by Zhang et al. [106] and Zhou et al. [108] optimize the communication costs of multiway joins in distributed systems. The approaches proposed by Gu et al. [42] and Aghbari et al. [10] focus on the distribution of loads on clusters to accelerate the processing, while others focus on the optimization

of multiway joins in sensor networks [110].

A multiway stream join operator named MJoin is presented in [95] based on a hashing technique. MJoin generalizes the symmetric binary hash join and the XJoin [91] algorithm for more than two input streams. Viglas et al. [95] demonstrate the benefits of MJoin over trees of binary joins, and Valsomatzis et al. [92] present a main-memory variant of MJoin that processes input data in batches with the objective of improving CPU efficiency. An extensive study of multiway stream joins in addition to performance improvements using various algorithms to maintain windows lazily in MJoin is presented in [70].

MJoin and, in general, multiway stream join solutions in software assume a communication model where any core can communicate with any other core, a model that cannot be directly realized in hardware, whereas our approach is based only on neighbor-to-neighbor communication.

Wang et al. [97] propose a multiway join operator distribution scheme called pipelined state partitioning (PSP). PSP partitions sliding windows (states) into disjoint slices in the time domain and then distributes the fine-grained states over a cluster, forming a virtual computation ring for processing incoming data streams. The proposed ring architecture defines various paths (each dedicated to a stream) between disjoint slices, resulting in an unstructured data path. Additionally, Wang's approach uses time-sliced windows, which require coordination to ensure state consistency, while our solution operates based on assigning an independent sliding window to each pipeline stage, thereby eliminating any need for coordination.

The caching mechanism presented by Babu et al. [16] adaptively allocates caches for pairs of streams based on a given order of join operators to improve processing throughput, which shares concepts similar to our Stashed-MJ. However, Badu et al.'s solution does not trivially map to a hardware realization due to the required flexibility to add and remove caches for various operators. In contrast, our approach uses a custom architecture that does not require expensive circuitry to achieve the required flexibility.

Work related to stream join acceleration can be broadly classified into software-based stream join algorithms [53, 39, 85, 43, 28, 107, 11, 94, 68] and hardware-based counterparts such as [100, 69, 67, 98, 56].

Gedik et al. [39] introduced the parallel CellJoin, designed for a heterogeneous architecture, which aims to substantially improve stream join performance by spreading tasks among processing elements to operate in parallel. However, CellJoin requires repartitioning through a central coordinator for each newly incoming tuple, which limits its scalability. The problem of distributed stream join processing has also been studied with respect to elasticity and memory reduction, which is applicable to cloud computing [43].

CellJoin's coordination challenge is partially addressed by the handshake join [85] which transforms the stream join into a data flow problem (evolved from Kang's three-step procedure [53]): tuples flow from left-to-right (for the first stream) and from right-to-left (for the second stream) while passing through each join core. Another approach, named SplitJoin, that similarly divides the sliding window into smaller subwindows is proposed in [68]. In contrast to [85], SplitJoin uses a unidirectional distribution mechanism (uni-flow) to transfer incoming tuples to independent processing (join) cores.

3.3 Hardware Stream Processor

Work related to our approach can be broadly classified into stream processing in software [23, 26, 7], approaches to performance-optimize stream processing through emerging hardware mechanisms [100], in particular, through FPGA-based acceleration [61, 44, 63, 84].

Additionally, since the inception of stream processing, the development of optimizations, both at the query-level and at the engine-level have been widely explored. For example, co-processor-based solutions utilizing GPUs [88] and more recently hardware-based solutions employing FPGAs have received attention [44, 84, 63, 99, 80, 81]. Below, we discuss these approaches and describe how our approach differs.

Software Approaches —

Stream processing has received much attention over the past few years. Many viable research prototypes and products have been developed, such as NiagaraCQ [26], TelegraphCQ [23], and Borealis [7], to just name a few.

Tumeo et al. showed how to use GPUs to accelerate regular expression-based stream processing language constructs [88]. The challenge in utilizing GPUs lies in transforming a given algorithm to use the highly parallel GPU architecture that has primarily been designed to perform high-throughput matrix computations and not, foremost, low latency processing. Also, the use of GPUs for acceleration suffers from similar memory wall and redundant memory access problems. The memory wall problem is even further exasperated in the case of GPUs due to additionally needing to transfer data from main memory to the GPU's memory. The use of FPGAs in our approach allows us to adapt and customize the underlying computing platform to match the algorithm, instead of having to work with a given architecture and programming model. This flexibility also gives us the freedom to exploit other application characteristics that a general purpose environment cannot offer.

Most existing systems are fully software-based, support a rich query language, but acceleration

through FPGAs has not been attempted in these approaches.

Aurora and Borealis. The Aurora and Borealis stream processing engines [22] make it possible to implement continuous queries by using a predefined set of operators that share concepts with those used here. By contrast, however, our approach targets hardware platforms with different criteria. Although the expected processing speedup and efficiency of a hardware solution are significant, a new system design is needed to use this approach on a hardware platform. Therefore, this work focuses on a detailed hardware design and relevant considerations.

The Borealis is a distributed stream processing system that inherits core stream processing functionality from Aurora and distribution functionality from Medusa (Brown) [17]. The Borealis modifies and extends both systems in nontrivial and critical ways to provide advanced capabilities that are commonly required by emerging stream processing applications.

STREAM. STREAM (Stanford) [15, 62] focuses on the development of a general-purpose data stream management system (DSMS) for processing continuous queries over multiple continuous data streams and stored relations similar to Aurora.

TelegraphCQ. Similar to previous approaches, TelegraphCQ [23] implements a stream processing engine. STREAM and TelegraphCQ both implement variants of CQL, the Continuous Query Language. CQL is a straightforward extension of SQL for manipulating streams where, instead of defining relations, we work with operators for streams.

CQL is the streaming-based version of SQL, with stream-relation and relation-stream operators rather than the relation-relation operators in SQL.

Gigascope. Gigascope [31] is a high-performance streaming database designed for monitoring networks with high-rate data streams using inexpensive processors. Because Gigascope is a stream-only database, it does not support stored relations or continuous queries. This restriction significantly simplifies and streamlines the implementation.

Hardware Approaches —

To further narrow our discussion, we focus on distributed real-time analytics over continuous data streams using FPGAs.

Past work showed that FPGAs are a viable option for accelerating certain data management tasks in general and stream processing in particular [44, 84, 63, 99, 80, 81].

Hagiescu et al. [44] identify compute intensive nodes in the query plan of a streaming computation. To increase performance, in the hardware design that realizes the streaming computation, these

nodes are replicated, which due to the stateless nature of the query language considered, poses few issues. An ulterior benefit is that design synthesis is sped up, as replicated nodes, are synthesized only once. A main difference to our work is the restriction to stateless operations and the lack to flexibly update the streaming computation.

Lockwood et al. [61] present an FPGA library to accelerate the development and verification of (financial) applications in hardware. Similarly, Mueller et al. [63] present Glacier, a component library and compiler, that compiles continuous queries into logic circuits on an operator-level basis. Both approaches are characterized by the goal of representing queries in logic circuits to achieve the best possible performance. While performance is a major design goal for us as well, we additionally aim at offering the application flexibility to update queries at run-time.

Sukhwani et al. [84] present a solution to offload compute intensive parts from an analytics query running on a standard CPU to an FPGA connected via the PCIe bus. Major speedups are due to moving compressed data between the compute elements via direct memory access (DMA). For latency sensitive processing, the approach is limited by the speed at which data can be copied from memory to the FPGA via the PCIe bus. While the use of DMA relieves the CPU from these copying tasks, the involved latency overhead cannot be avoided. In comparison to our approach, Sukhwani et al. [84] aim to accelerate traditional query-retrieve-style processing, unlike the streaming query model that we aim to support.

Streams on Wires. Hardware solutions are known for their inflexibility to updates. To address this, the concept of Streams on Wires [63] assumes that the set of queries is known in advance. Past work has relied on the compilation of static queries and fixed-stream schema onto hardware designs synthesized to configure FPGAs. Depending on the complexity of the design, this synthesis step can last minutes or even hours. This is not suitable for the needs of modern-day stream processing: fast and on-the-fly reconfiguration. Furthermore, prevalent approaches assume a complete halt to the hardware reconfiguration phase [63, 78]. This requires additional logic for buffering, handling of dropped tuples, requests for retransmissions, and additional dataflow-controlling tasks, which renders this style of processing difficult in practice.

SPARC M7. Oracle SPARC M7 [12] is a commercial chip that contains stream-processing hardware accessed through the DAX API. Using this API allows access to this hardware using common software commands to efficiently manipulate and process large amounts of data. Units such as this are not directly comparable with our solution because they use a fixed architecture customized for specific tasks (e.g. data filtering).

Q100. Wu et al. in [101] proposed a domain-specific database processor named Q100 to efficiently handle database applications. The Q100 contains a heterogeneous collection of fixed-function ASIC tiles, each of which implements a well-known relational operator, such as a join or sort.

In contrast to Q100, which is built for database operations, our solution is custom designed for stream processing, which fundamentally changes the hardware architecture design requirements. Our solution is designed to support various query types in the stream processing and its framework allows the rearrangement of components and utilization of almost any type of processing component.

With regard to similar approaches, we can refer to the IBM Netezza [36], which is a commercial product that exploits parameterizable circuits to offload query computation. The IBM Netezza is a parallelized system that is purpose-built for data warehousing. It is commonly referred to as a data warehouse appliance designed specifically for running complex data warehousing workloads. The concept of an appliance is realized by integrating the database, server, and storage into a system that is easy to deploy and manage. In any database system, the main bottleneck is input/output. The Netezza reduces this bottleneck by using a commodity FPGA by pushing the SQL closer to the silicon to help improve input/output ports performance. This core component of the appliance is referred to as the database accelerator.

The IBM Netezza targets many aspects of data handling and processing centered on databases and warehousing. Our porposed solution can be complementary, and is usable as a component in the Netezza to handle streaming data workloads with the additional coarse-grained and fine-grained flexibilities provided in our solution.

Utilizing general purpose soft-cores on FPGA may also seem like a viable alternative. However, soft-cores are meant to support complex computing tasks that are too expensive to be realized in hardware and are not computationally intensive to become a system bottleneck. A comparison along these lines for event processing operations showed that general purpose soft-cores are not interesting from performance perspective, at least in the targeted context [77].

Chapter 4

Configurable Stream Processor

The limitations of traditional general-purpose processors have motivated the use of specialized hardware solutions (e.g., FPGAs) to achieve higher performance in stream processing. However, state-of-the-art hardware-only solutions have limited support to adapt to changes in the query workload.

In this work, we present a reconfigurable hardware-based streaming architecture that offers the flexibility to accept new queries and to change existing ones without the need for expensive hardware reconfiguration. We introduce the *Online Programmable Block* (OP-Block), a "Lego-like" connectable stream processing element, for constructing a custom Flexible Query Processor (FQP), suitable to a wide range of data streaming applications, including real-time data analytics, information filtering, intrusion detection, algorithmic trading, targeted advertising, and complex event processing.

Through evaluations, we conclude that updating OP-Blocks to support new queries takes on the order of nano to micro-seconds (e.g., 40 ns to realize a join operator on an OP-Block), a feature critical to support of streaming applications on FPGAs.

This section presents the following contributions:

1. The design of a parameterizable stream processing element called *Online Programmable-Block* (OP-Block) that lies at the core of a hardware-based Flexible Query Processor (FQP). Our design enables a streaming architecture by decomposing and implementing complex queries into simpler OP-Block components. Through hardware specialization and dividing chip area into many small, self-contained, independent OP-Block instantiations, we eliminate the need for large and overly complex control units for coordination, and achieve better chip area utilization.

2. We enable processor and memory coupling within the OP-Block that supports direct I/O access to provide an explicit link between I/O ports and processing unit(s) within the OP-Block, which reduces problems that are due to the *memory wall* and *redundant memory accesses*.

3. We support dedicated instruction memory within the OP-Block that provides low latency access to both queries and data, and avoids complex and expensive instruction fetching that are common for moving data between CPUs and external memory in today's computing architecture, thereby, addressing the *von Neumann bottleneck*.

4. We enable the fine-grained dynamic reconfiguration of the OP-Block that allows the application to re-program it in an online fashion without disrupting query processing.

5. We enable the coarse-grained static reconfiguration of the OP-Block inter-connection topology that allows us to attach and connect several OP-Blocks to realize complex queries and to adapt to the specific processing needs of these queries.

6. We support variable tuple sizes by proposing the segment-at-a-time processing model, namely, an abstraction that divides a tuple into smaller chunks that are streamed and processed as a consecutive set of segments. This strategy avoids the need for over-provisioning of hardware resources.

4.1 Stream Processor Design

In this section, we describe the design of our custom stream processing element, called OP-Block. We describe its instruction set, internal architecture, design rationale, and present an example of how queries are programmed and executed.

4.1.1 OP-Block Design Overview

We designed the *Online Programmable-Block* (OP-Block) specifically tailored to the processing characteristics of data streams, such as shared processing of a set of queries over numerous tuples, reduction of redundant memory accesses due to streaming nature of I/O and processing, and the need to sustain predicable processing latency for high tuple rates.

Moreover, our design has been heavily influenced by the requirements of the stream join operator, as it is often the most involved query operator. In its basic form, the join operator consumes two input streams and produces one output stream. To maximize the input tuple rate and the processing throughput, each stream is supported by a dedicated port and a window buffer, realized by non-shared BRAM memory in the FPGA.

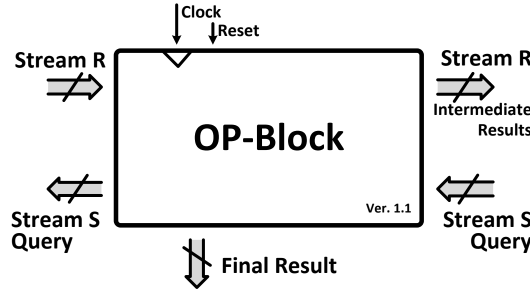


Figure 4.1.1: OP-Block: I/O ports.

The design of the OP-Block, including its I/O ports, is shown in Figure 4.1.1. The width of each port is equal to the size of a single tuple in addition to bits reserved for auxiliary signals. One input stream is fed to the OP-Block from the upper-left port (Stream R), while the other input stream and queries enter from the lower-right port (Stream S/Query). For continued processing, tuples in window buffers are emitted from the upper-right and lower-left ports, while intermediate results are always emitted from the upper-right port. The final processing result, upon completion of the query, is emitted from the lower port (*Final Result*), where the processing results become available.

OP-Blocks can be connected via their left and right I/O ports. We refer to such a configuration as a pipeline since each OP-Block behaves similar to a pipeline stage in the data stream processing. In this way, join operators with large window buffers or query plans with data dependency between operators, that cannot be processed by a single OP-Block, can be mapped across multiple OP-Blocks. In addition, OP-Blocks can be arranged in a parallel manner serving to support independent query plans (e.g., query plans comprised of multiple queries, not sharing intermediate results, thus, amenable for parallel processing.) Furthermore, hybrid constellations of OP-Blocks combining pipelining and parallelism, for example, are an option for building custom stream processing architectures that are specifically tailored to given query plans.

4.1.2 Instruction Format

Before presenting our detailed design, we describe the instruction format, which defines the instruction set and data layout of tuples the OP-Block understands. A query, its operators, and stream tuples are translated into the instruction set and data format. Below, we use the terms instruction and query/operator interchangeably, referring to instruction when describing stream processor internals.

Our OP-Block design is configurable via a number of synthesis-time parameters such as: attribute and value widths (in bits), number of attribute-value pairs, and size of internal buffers, including window buffers for the join operator. Assuming a tuple size of 64-bit (32-bit attribute and 32-bit

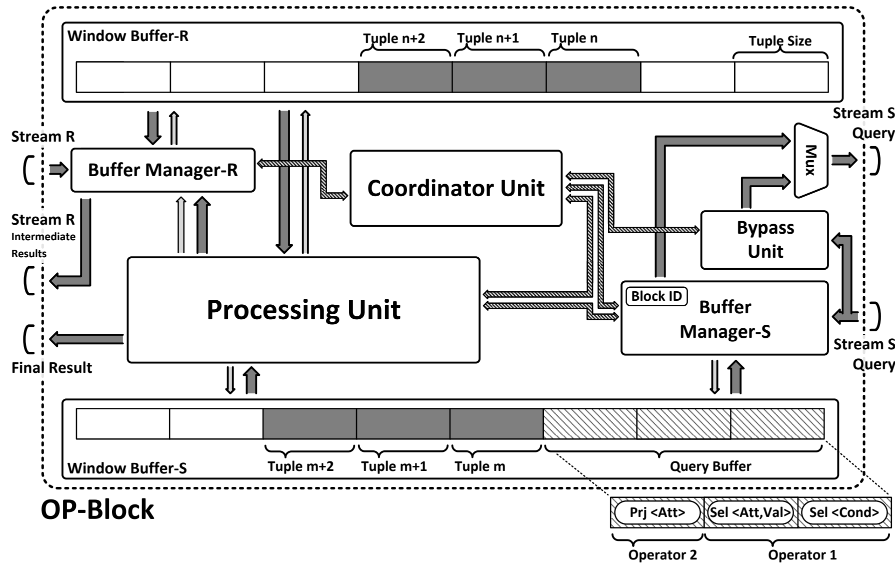


Figure 4.1.2: OP-Block: internal architecture.

value), our instruction and data format is shown in Figure 4.1.4. In the figure, bold dashed lines represent bytes and faint dashed lines represent bits in a byte. We call each sequence of nine consecutive bytes a *chunk*. A tuple is comprised of one chunk.

The *selection*, *projection*, and *join* operators together with a *reset* instruction are identified by the `Inst` bits (cf. Figure 4.1.4). A single OP-Block is capable of supporting multiple instructions and can be used to implement selection, projection, and join operators.

The first byte of each chunk contains two bits: S and Q. The S bit specifies the data stream a tuple belongs to, either zero, for Stream R, or one, for Stream S. The Q bit differentiates between the type of input, either zero, a tuple, or one, an instruction (i.e., a query operator). The remaining parts in a tuple are the attribute and value fields, in byte positions 2-5 and 6-9, respectively. Each instruction is targeted at a specific OP-Block, among the existing blocks determined by the `Block ID` (B-ID) field. In addition, a `Chunk Count` (CC) specifies the number of chunks per instruction. For example, a 2-bit `Inst` and 4-bit `B-ID` field are sufficient to discern 4 instructions and 16 OP-Blocks, respectively; by tuning these simple design parameters, additional instructions and OP-Blocks can be supported.

The selection operator is comprised of an attribute, a value, and a condition field (i.e., encapsulated in three chunks), the join operator requires an attribute and a condition field (i.e., two chunks), and the projection operator requires an attribute field (i.e., two chunks). The *reset* instruction serves to initialize an OP-Block. By using the `Policies` field, *reset* selects the stream result output port for a given query to process.

4.1.3 Architecture

The OP-Block is comprised of hardware components working in parallel to maximize processing performance. The OP-Block's internal architecture is shown in Figure 5.5.1. Some component names carry the postfix -*R* and -*S*, which indicates that they are handling either *Stream R* or *Stream S*, respectively.

Window Buffer (-R&-S) is a memory unit which realizes the sliding window concept for a join operator. The buffer is comprised of a synthesis-time definable parameter for setting the maximum number of tuples for each buffer (at runtime for larger buffer sizes, multiple OP-Blocks can be connected). This buffer also serves to store query operator parameters such as list of projected attributes, selection conditions, and join conditions (cf. *Query Buffer* in Figure 5.5.1). The memory is implemented with BRAMs, the dedicated high-performance memory of the FPGA. Essentially, the window buffer acts as low-latency, on-chip data and instruction cache that is embedded within each OP-Block.

Buffer Manager-R handles Stream *R* in addition to forwarding intermediate results to neighboring blocks connected to its output port.[1] This component tracks the status of its associated window buffer (i.e., full vs. empty), and based on requests on the OP-Block's input port and grants from the *Coordinator Unit*, pushes and pops tuples to and from *Window Buffer-R*. To maximize parallelism, this component has two independent reception and transmission controllers. The *reception controller* is responsible to retrieve tuples from Stream *R* and inform other units about a tuple newly inserted into *Window Buffer-R*. The *transmission controller* sends out tuples available in the window buffer, including intermediate result tuples, through the Stream *R* output port.

Buffer Manager-S is similar to *Buffer Manager-R*, except that it additionally handles instructions arriving at the Stream *S* input port. In case an instruction is targeted at the current OP-Block the *reception controller* inserts it into the instruction region of *Window Buffer-S* (cf. Figure 5.5.1.) This region is separated from the rest of the buffer window (i.e., the region for tuples) and expands as more instructions are inserted into the OP-Block.

The *Bypass Unit* acts as tuple relay mechanism that enables queries and tuples to flow through the OP-Block topology. Recall that OP-Blocks are inter-connected based on a synthesis-time defined topology, and OP-Blocks may be assigned at runtime to different queries, which is the key to our coarse- and fine-grained flexible query processing architecture. For example, to get a tuple to the right query, and subsequently to the right OP-Block, we need a relay mechanism to move the tuple from a source (an OP-Block) to a destination (another OP-Block), where the path from the source to the destination may span many intermediate OP-Blocks. All the intermediate OP-Blocks act as bypass for the sake of moving the tuple to its destination.

[1] Intermediate results are tuples resulting from one operator fed, as input, to another operator in the query plan.

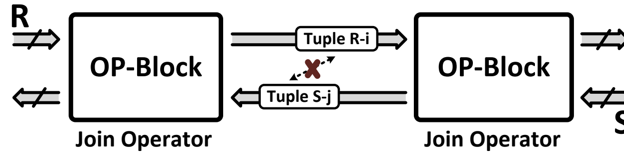


Figure 4.1.3: Simultaneous tuple transmission issue.

Thus, the *Bypass Unit* passes the input from the OP-Block Stream S input to its Stream S output port, provided the input is targeted at subsequent blocks. This component has a small internal buffer which is managed by two independent controllers that read/write incoming instructions from/to the buffer. The buffer is required to facilitate the transmission of tuples from the input to the output port without involving other OP-Block internal units.

In case the instruction available at the Stream S input port does not belong to the current OP-Block, the write controller sends a request, for a bypass operation to the *Coordinator Unit*. After receiving a grant signal, the write controller reads the CC field of the instruction to determine the number of chunks to bypass, which is stored in the first chunk of each instruction. Subsequently, the write controller continues to store the rest of the instruction chunks in the internal buffer, in addition to the first chunk. Simultaneously, the read controller sends a request for transmission to its neighboring block connected to its Stream S output port, as long as its internal buffer is not empty.

The *Coordinator Unit* controls permissions and priorities to manage data transmission requests to ensure query processing correctness. For example, one important rule dictates that on each side of an OP-Block, only one of the reception and transmission controllers is allowed to operate at each time interval. In other words, *Buffer Manager-R* reception and *Buffer Manager-S* transmission controllers or *Buffer Manager-R* transmission and *Buffer Manager-S* reception controllers are not allowed to work simultaneously. This rule prevents the simultaneous transmission and reception of tuples on each side of the block. Therefore, by enforcing that only one tuple is moved at a time between two neighboring OP-Blocks greatly simplifies the control logic and the overall design. The possibly erroneous outcome when not following this rule is shown in Figure 4.1.3. Another rule prohibits changes to the contents of window buffers, while the the *Processing Unit* is running. This simplification applies only to the window buffer whose change in content affects the resulting output stream.

The *Processing Unit* (PU) is the execution core of the OP-Block. It is activated when a new tuple is inserted. Then, based on the programmed queries, it executes every instruction in the *Query Buffer* on the new tuple. For example, assume the OP-Block is programmed with two selection operators and a new tuple enters, then the PU executes the corresponding instructions over the tuple in consecutive steps. In case of a join operator, this unit joins the new tuple with every available tuple in the opposite window buffer. Our current implementation is based on the nested loop join algorithm.

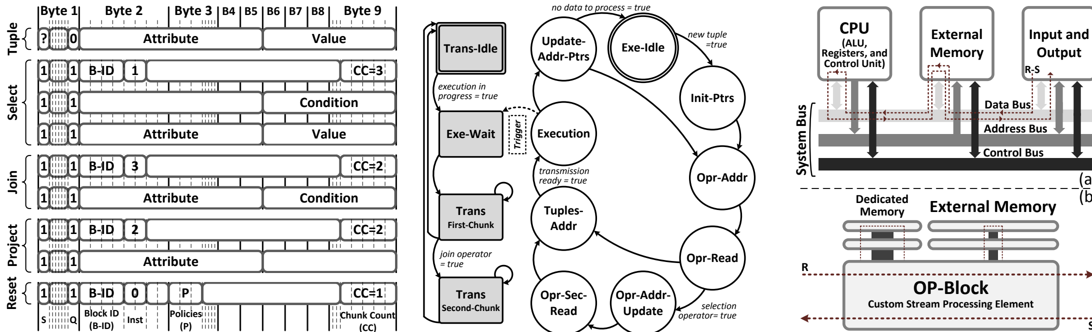


Figure 4.1.4: Instruction and data format.

Figure 4.1.5: PU state diagrams.

Figure 4.1.6: (a) System bus (b) OP-Block.

The PU has an *execution* and a *transmission control unit.* The *execution unit* fetches and executes instructions on tuples, while the *transmission unit* outputs the query result tuples. The state diagrams for the controllers of both units are shown in Figure 4.1.5: After receiving a new tuple, the execution controller determines in the `Opr-Addr` state, the address from where to fetch the first operator from within *Window Buffer-S*. Then, it reads it in state `Opr-Read`. For a selection, an extra step is required to read the third instruction chunk, which is done in state `Opr-Sec-Read`.

In the `Tuples-Addr` and the `Execution` states, the addresses of tuples to process are calculated and the tuples are read from either one or both of the window buffers. Actual processing is done in the `Execution` state. Finally, addresses for the remaining instructions and tuples are updated for the next iteration in `Update-Addr-Ptrs` state. When there are no more operations to perform, the execution controller terminates in state `Exe-Idle`.

As the execution controller starts processing, the transmission controller waits (in `Exe-Wait` state) for result tuples that it can pass on. In the `Execution` state, the execution controller triggers the transmission controller, where the transmission controller then emits the result tuples in `Trans First-Chunk` and `Trans Second-Chunk` states, respectively. `Trans Second-Chunk` is utilized in the case of a join operator where the resulting tuple is the concatenation of the two input tuples.

Finally, the *Filter Unit* is attached to the *Final Result* output port of the OP-Block as shown in Figure 4.2.2. It serves as an output buffer, which mitigates stalls imposed by erratic saturation (i.e., a temporal increase in the match rate) in output buffer queues.

4.1.4 Operation Example

Next, we describe how a query is programmed and executed via a number of OP-Blocks. Consider the following query (Figure 4.1.7) that filters out customers with age less than or equal 25.

```
CREATE STREAM CS_SEL
AS
SELECT *
  FROM
Customer_Stream
 WHERE Age > 25

CREATE STREAM
CS_PROJ AS
SELECT Name, Price
  FROM CS_SEL
```

Figure 4.1.7: SQL code example of a query.

To apply this query to an OP-Block first, we issue a reset instruction to configure the OP-Block output port. Second, we issue a selection instruction that contains selection operator fields (i.e., condition, attribute, and value). Finally, we issue a projection instruction to project out undesired fields.

To apply the selection operator, *Buffer Manager-S* within the OP-Block receives instructions that carry the selection condition and writes them into the *Query Buffer*.

During processing, `Customer` input tuples enter the OP-Block from the Stream R input port. Each tuple in the `Customer` stream is handled by the *Buffer Manager-R* and is placed in *Window Buffer-R*. Next, the *Processing Unit* is notified, which then executes the selection and the projection operators on the newly entered tuple.

In general, a selection, projection, or join operator is executed by an OP-Block in four steps:

1. Arriving tuple is stored in one of the two window buffers.
2. Depending on the operator one or more tuples are fetched by the PU from the window buffers.
3. The actual operation is applied to the tuples accordingly.
4. The resulting tuples (if any) are transferred out via *Final Result* or Stream S output port.

Issuing a query involving a join is similar to projection and selection. However, for joins, two input streams, Stream R and Stream S, are handled at the input ports.

4.1.5 Design Rationale

Here, we summarize how the characteristics of stream processing influenced the design of the OP-Block as well as how and why we were able to address some of the well-known disadvantages of today's computer architectures.

Design for Parallelism and Pipelining —

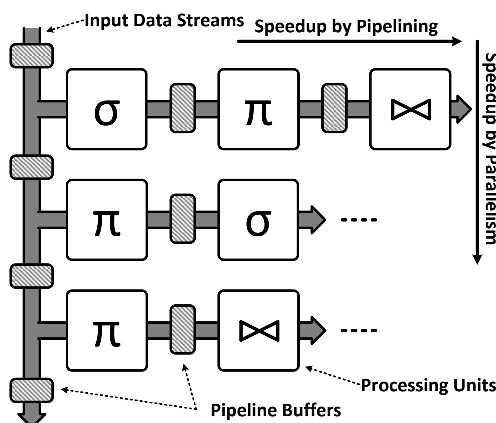


Figure 4.1.8: Parallelism and pipelining arrangements.

Each OP-Block is self-contained and a set of OP-Blocks can be inter-connected to form a query-specific topology. Each arrangement (cf. Figure 4.1.8) can be optimized with respect to two different design criteria, namely, parallelism and pipelining. Consider the stream processor as a grid of connected OP-Blocks. The width of the OP-Block grid dictates the degree of available pipelining for complex queries that have large inter-dependency pipelines. The length of an OP-Block grid controls the level of inter- and intra-query parallelism, in which there is no data dependencies among the OP-Blocks (i.e., operations are not blocking).

The goal of our architecture is to provide the necessary framework to allow query assignment to determine an optimal topology given statistics about the workload. The framework supports the mapping of a logical OP-Block configuration (i.e., logical query plan) to a fixed OP-Block grid (i.e., physical query plan).

Design for Direct I/O Access —

Stream processing applications are often characterized by high I/O rates [37]. In our design, OP-Blocks are directly connected to the system's I/O ports without requiring redirecting the data flow through main memory, thus, alleviating the memory wall issue and avoiding redundant memory accesses.

The redundant memory access problem is a major source of memory bandwidth consumption in today's computer architecture: Input arriving from an I/O device is first written to memory (initiated by the CPU) and subsequently, for processing, the CPU reads the data again from main memory to process it. Output is processed analogously. As a result, read and write from and to I/O sum up to at least two redundant memory accesses that are remarkably expensive in the face of processing potentially unbounded and high-rate streams, especially, if the executed queries do not build up much state or are entirely stateless. Figure 4.1.6(a) and (b) show the difference between

the data path a stream takes from input to output in the conventional system bus and the OP-Block architecture, respectively.

In our design, we use the FPGA's on-chip BRAMs, which are distributed across the chip and tightly coupled with the OP-Block. There is no need for shared memory between OP-Blocks, which contributes to the scalability of FQP by avoiding memory access coordination and coherency issues. Within an OP-Block, BRAMs serve as local memory elements, such as internal buffers to decouple OP-Block elements and to realize window buffers for the join operator.

Design for High Performance/Transistor-Count Ratio —

The focus on building custom hardware, only tailored to stream processing, allows us to drastically simplify the design of the OP-Block by eliminating and simplifying the central coordination and control logic without which general purpose processors would not function. For example, no sophisticated (central) memory system is required between input ports and core processing units. Relatively speaking, the net result is that fewer chip resources have to be devoted to control units and cache units, respectively. This leaves more resources for the OP-Block's *Processing Unit*. For example, in our design more than 60% of the chip resources make up the PU.

Design for Flexibility —

Related approaches, dedicated to accelerating stream processing through FPGAs, require lengthy re-synthesis in face of supporting new or updated queries in their designs. Also, apart from needing to modify queries online, data streams tend to evolve (e.g., arrival distribution variations and data format changes) over time [37]. Our design features *online programmability* as its main and distinct characteristics, as compared to the state-of-art in accelerating stream processing with FPGAs. Also, in our design, an OP-Block features an instruction memory (cf. *Query Buffer* in Figure 5.5.1). Query operators reside in this memory, which can be updated at runtime to modify the executed operator and query. Our design avoids the von Neumann bottleneck, as it keeps instructions inside the processing core, the OP-Block. Essentially, we realize the concept of *processing data (streams) over instructions*, not the conventional model of executing instructions over data, where both reside in a deep cache hierarchy.

Design for Simplicity —

To overcome issues relating to design complexity, we divide our hardware design into an intra- and inter-OP-Block design. The intra-OP-Block design relates to the internal architecture of an OP-Block and includes instructions set, port configurations, input/output data format, and design of internal components like *Coordinator* and *Processing Unit*. The inter-OP-Block design specifies how multiple OP-Blocks are inter-connected to each other to build a stream processor. For example, OP-Blocks can be instantiated serially (in a chain cf. Figure 4.2.2) to realize operators in a query plan or to build

```
CREATE STREAM CS_SEL1 AS
SELECT *
  FROM Customer_Stream
 WHERE Age > 25;

CREATE STREAM CS_OUT1 AS
SELECT *
  FROM CS_SEL1 [Rows 1536], Product_Streams [Rows 1536]
 WHERE Order ID = Product ID;

CREATE STREAM CS_SEL2 AS
SELECT *
  FROM CS_SEL1
 WHERE Gender = female;

CREATE STREAM CS_OUT2 AS
SELECT *
  FROM CS_SEL2 [Rows 2048], Product_Streams [Rows 2048]
 WHERE Order ID = Product ID;
```

Figure 4.2.1: SQL code example of a query.

a join operator with large window buffers. This results in a performance gain by adding processing pipeline stages as illustrate in Figure 4.1.8. OP-Blocks can also be instantiated in parallel to realize independent query plans which increase performance by adding processing units in parallel, as shown in Figure 4.1.8.

Design for Fine- and Coarse-Grained Configurability —

The inter-OP-Block design enables building a custom topology for a specific stream computation. For example, one computation may require high filtering throughput while another one may aim to support complex join correlations between large tuple windows. The former can benefit from using many parallel blocks since stateless filtering is easily separable, while the latter, due to data dependencies, has to be processed more serially, obtaining speedup from pipelining. Consequently, a single architecture for all stream computations is not the best option, rather a configurable topology is what is needed. On the one hand, our design supports fine-grained configuration by allowing the application to insert queries at runtime and, on the other hand, course-grained configuration by allowing to custom assemble a new instance of a stream processor that supports an altogether different set of stream computation requirements.

4.2 FQP Configuration Examples

In this section, we exemplify how multiple OP-Blocks can be configured into different inter-connection topologies to realize a given query on the FQP. We discuss pre-processing steps that prepare the input for processing. We also illustrate how queries are mapped onto the processor's configuration — a step referred to as *query assignment*. We qualitatively discuss the trade-offs that result from different OP-Block configurations realizing the same query. A quantitative evaluation is presented in Section 4.5 and a formal treatment of query assignment is deferred to future work.

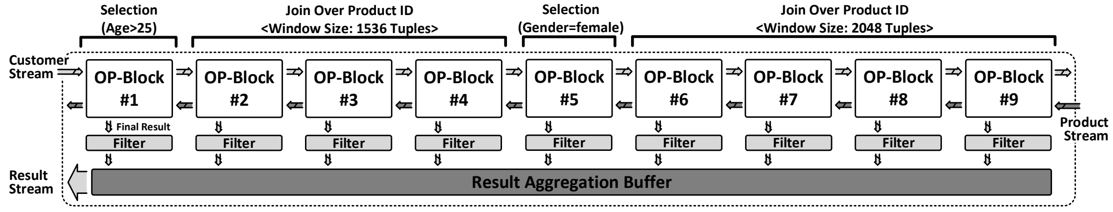


Figure 4.2.2: Query assignment in OP-Chain topology.

During pre-processing, queries and streams are transformed into the appropriate input format, — the instruction format described in Section 4.1.2, — understandable by the stream processor. A single query may map to multiple instructions executed by the processor.

In our below example, the query (cf. Figure 4.2.1) filters out customers who bought a specific product and are older than 25 over a window of size 1536 (i.e., 512×3) and also customers who are female in addition to being older than 25 over a window of size 2048 (512×4).

Also, during pre-processing, in a step resembling query compilation, the instructions representing the query plan are assigned to appropriate blocks of the FQP configuration. We refer to this step as *query assignment*, which receives queries and block topology as input and, based on the location of each operator in the query plan and positions of OP-Blocks in the topology, assigns each operator (or set of operators) to a specific OP-Block.

The query assignment is a critical step in efficiently utilizing the statically allocated hardware resources (i.e., the block topology) with regards to latency, throughput, and power consumption. For example, to reduce latency, a selection operator, should be assigned to an OP-Block closets to the processor's entry point. As a result, query assignment may trigger the reprogramming of other OP-Blocks to free up specific blocks in favor of overall processing performance. In contrast, a poorly chosen query assignment may increase query execution latency, leave some blocks un-utilized, negatively affecting energy use, and degrade the overall processing performance. Generally speaking, the query assignment complexity increases with the size of the query and the number of OP-Blocks.

We now illustrate the assignment task, first, given a chain of OP-Blocks as underlying processing topology and, later, a parallel configuration.

OP-Chain topology — A chain of OP-Blocks (OP-Chain), inter-connected in a pipelined manner, is shown in Figure 4.2.2, where queries are processed against `Customer` and `Product` streams entering the chain. Stream processing results are collected from individual OP-Blocks and are guided to FQP's output via the *Result Aggregation Buffer*, which is a result gathering tree-like structure with buffers in its internal nodes.

We start the assignment of the SQL query from Figure 4.2.1 to the OP-Chain in Figure 4.2.2

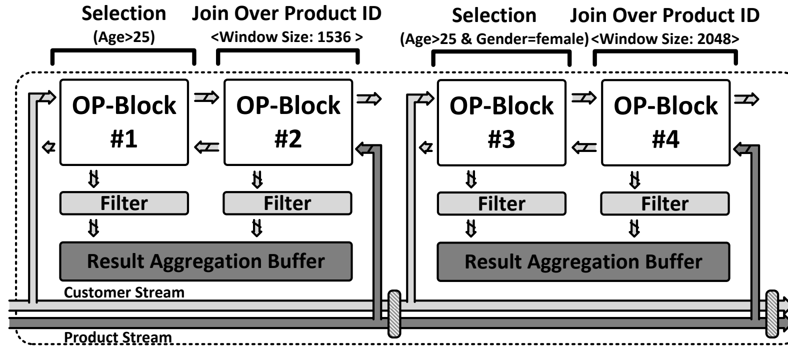


Figure 4.2.3: Query assignment in partial parallel topology.

by mapping the selection operator on `age` to OP-Block-#1 and program it by a reset and a select instruction. The reset instruction sets the OP-Block-#1 result port to thr *Stream R* output port, where the selected tuples are passed on to OP-Block-#1's right neighbor for subsequent join processing. Both instructions are inserted into the chain through OP-Block-#9's Stream S input port, where, by carrying the identifier of their target OP-Block, they bypass other OP-Blocks, before reaching the target block.

We can realize join operators with various window sizes equal to multiples of the OP-Blocks' window sizes. In this example, each OP-Block has a window size of 512 tuples; therefore, three OP-Blocks are required to realize the join operator in Figure 4.2.1 with window size 1536 (3×512). To apply the join operator, two instructions, for each of OP-Blocks-#(2-4), are issued. Again, reset is the first instruction, but in contrast to the previous operator, it sets the result port to be the *Final Result* port in OP-Blocks-#(2-4), where tuples are gathered by the *Result Aggregation Buffer* for transfer as processor output.

The selection operator on `gender` is mapped to OP-Block-#5 and the remaining join operator is mapped to OP-Blocks-#(6-9). To realize a window size of 2048 (4×512) four OP-Blocks are needed.

In stream processing mode, the stream processor receives `Customer` tuples from the Stream R input port of OP-Block-#1, applies the selection operator, and emits filtered tuples as intermediate result stream through its Stream R output port. Furthermore, the processor receives `Product` tuples via the Stream S input port of OP-Block-#9 and produces the result streams at *Final Result* port of OP-Blocks-#(2-4) & #(6-9), respectively. Note that the second selection operator is applied at OP-Block-#5 and the intermediate result stream is sent to OP-Block-#6 from Stream R's output port of OP-Block-#5.

Partially Parallel Topology —

In the query assignment for the OP-Chain topology, we assigned the join operators to multiple OP-

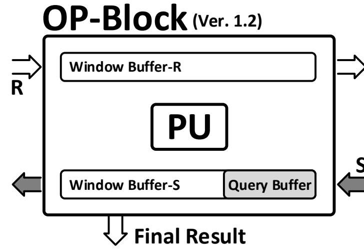


Figure 4.3.1: Stream processing element.

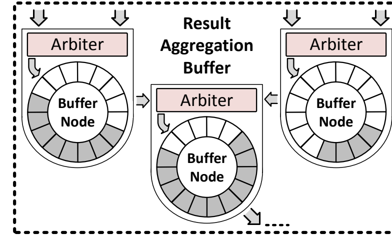


Figure 4.3.2: Result aggregation buffer (RAB).

Blocks to accelerate the processing by benefiting from pipelining. Figure 4.2.3 shows the assignment of the same query to fewer OP-Blocks arranged in a partially parallel topology: OP-Blocks-#1 & #2 and OP-Blocks-#3 & #4 are each pipelined and parallel to each other. Since this assignment does not allow for data dependencies between parallel parts. The query is duplicated at the first selection operator, `Selection(Age>25)`, repeating this operator in both parallel branches. The rest of the query assignment is similar to the one in the OP-Chain.

One advantage of the parallel topology is that it requires fewer processing cycles for streams to reach the corresponding OP-Blocks. Thus, overall, it enables lower latency processing. For example, the `Product` stream in the OP-Chain configuration has to pass through four OP-Blocks bypassing one OP-Block to reach OP-Block-#4 (the first join operator), while in the parallel configuration, the stream is directly injected into the first join operator.

In the configuration in Figure 4.2.3, OP-Blocks-#2 & #4 have window sizes of 1536 and 2048, respectively. However, there is no limit in the number of OP-Blocks and their configurations; instantiating more OP-Blocks to benefit from pipelining is a viable alternative.

4.3 Segment-at-a-time Processing

Not only queries change throughout the life of an application, but the streams themselves evolve as well. Their properties such as schema, tuple size, and input rate change continuously. These features are at odds with today's FPGA-based stream processing solutions, which have, for the most part, been tailored to process one specific tuple width before requiring re-synthesis if tuple size changes are permitted at all. This degree of flexibility poses a severe challenge for a hardware-based solution, as opposed to its software counter-part. Our design has been specifically built to afford this flexibility. The OP-Block-based design of FQP supports varying size tuples, thus, allowing for evolving data streams.

Generally speaking, hardware systems have fixed size input ports, internal communication buses,

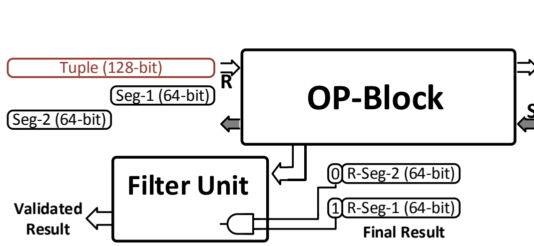


Figure 4.3.3: Segment-at-a-time at entry to OP-Block.

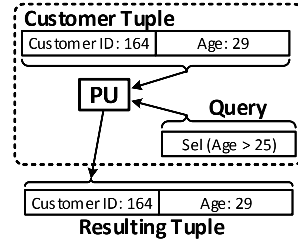


Figure 4.3.4: Customer segregation query.

and output ports. FQP is no exception. However, flexibility in the face of varying size data streams stems from the way an OP-Block processes incoming tuples. The parametrized design of the OP-Block allows us to define its ports' width prior to design synthesis. By default, we configure FQP with a 64-bit port width. As a result, for any tuple larger than 64 bits, it is divided into 64-bit segments at the entry-point to FQP. The tuple segments arrive at the input port of an OP-Block as shown in Figure 4.3.3. Then, the OP-Block processes each segment, one at a time, and hands over the resulting segments to the *Filter Unit* through its *Final Result* port.

Figure 4.4.1 shows the segment-at-a-time processing mechanism in more details. Prior to processing a segmented tuple, queries also need to be updated to handle the segments. In our example, the query consists of two segments, of which the first segment corresponds to the first segment of the tuple, while the second segment of the query corresponds to the second segment of the tuple. Segmentation of queries is performed in software outside of FQP.

The PU fetches the first segment of the tuple from the *Window Buffer-R* as well as the first segment of the query. Then, the PU executes the segment of the query and produces a result segment with an additional flag which shows if the first segment of the tuple satisfies the (query) conditions in the first segment of the query. This process is repeated for the second segment of the tuple etc.

All resulting tuple segments are transmitted to and stored in internal buffers of the *Filter Unit* (FU), which evaluates the validity of the entire resulting tuple. For example, in a selection operator, one of the tuple segments may not pass the selection condition, while others do[2], which would render the entire tuple invalid. After receiving the final segment and positively validating the result, the FU hands the tuple (a segment at a time) over to the RAB to transfer it to the output port of FQP. Otherwise, the FU drops all result segments.

Segment-at-a-Time Processing for Join Operator —

Query assignment is a task performed in software that maps the input queries onto the available blocks of the FQP configuration. This task determines the placement of operators, which is not known a priori (i.e., we do not know where a join operator executes). Segment-at-a-time processing

[2]E.g., the higher order bits pass the condition, while the lower order ones do not (for two segments).

is necessary to support the further processing of tuples that result from a join operation as often, the join result is comprised of both input tuples (unless attributes are projected out).

Segment-at-a-Time Tuple Size Limit —

The maximally accepted tuple size is determined by the size of *Window Buffer-(R&L)* in the OP-Block. From a conceptual point of view, the size of *Window Buffer-R* is not limited for stateless operators (i.e., select and project), while this is not the case for stateful operators (i.e., join). The actual limit depends on the resources available on the FPGA, which is highly device-specific and will only increase in future FPGAs. With today's technology, we have synthesized blocks with window sizes of up to 4K bytes.

4.4 Variable Tuple Size Example

Here, we give an example to illustrate the segment-at-a-time mechanism realized by the OP-Blocks. Assume a `Customer` stream with `Customer ID` and `Age` fields.

```
CREATE STREAM CS_SEL
AS
SELECT *
  FROM
Customer_Stream
 WHERE Age > 25
```

Furthermore, assume a query to segregate customers into two groups, those who are older and those who are younger than 25 years of age (e.g., a retailer wanting to compute recommendations based on age.)

This query is programmed onto the OP-Block and executed over the customer tuples as shown in Figure 4.3.4. As the `Customer` stream evolves over time, new attributes, such as `Height` and `Weight`, are added (e.g., for the retailer to better differentiate recommendations.)

```
CREATE STREAM CS_SEL AS
SELECT *
  FROM Customer_Stream
 WHERE Age > 25, Height <
180
```

Thus, the query is re-written as follows and through the segment-at-a-time mechanism, the OP-Block can execute the new query over the larger tuples without any changes as shown in Figure 4.4.2.

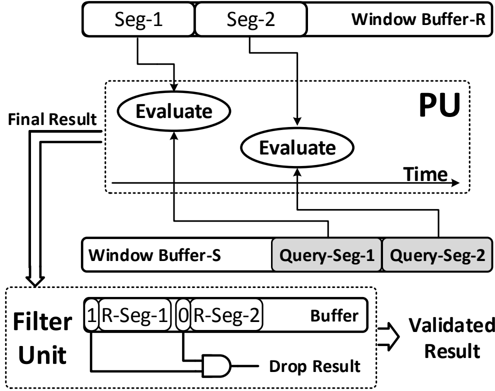


Figure 4.4.1: OP-Block Segment-at-a-time.

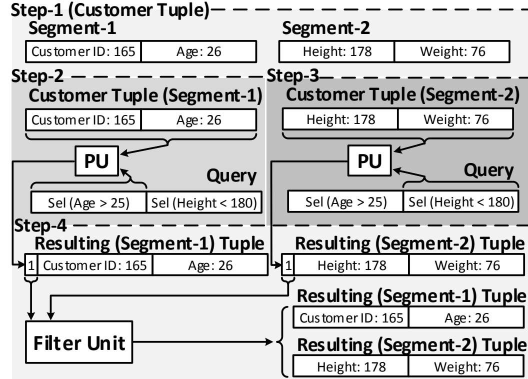


Figure 4.4.2: Segregation query.

Table 4.5.1: OP-Chain synthesis parameters.

	Buffer size				OP-Block	
Figure(#)	R-Pipe	T-Pipe	Filter-U	Buffer-N	Count	Win-Buf
4.5.2	20	20	4	4	16	4-256
4.5.7	20	20	4	4	16	8-16
4.5.3 & 4.5.4	4	4	4	4	2-28	4-4K
4.5.6	4	4	4	4	2-28	8
4.5.5	-	-	2	2	2-512	8

The processing of the updated (larger) tuple is done in four steps. In Step 1, after the query for the updated tuple schema was re-programmed onto its (target) OP-Block, the updated (enlarged) tuple is divided into two segments at the entry point of the FQP. In Step 2, that is, after the segments arrive at the target OP-Block, the *Processing Unit* fetches the first part of the query (Age $>$ 25) and executes it on the first segment of the tuple. In Step 3, the same process is repeated for the second part of the query (Height $<$ 180) and the second segment of the tuple. Each one of these steps produces a resulting tuple segment together with a validation flag. Finally, in Step 4, the resulting tuple segments are processed jointly using the *Filter Unit*. In case all segments have satisfied the query conditions, they are handed over to the RAB for transfer to the output port of the FQP. In this example, for illustration purposes, we have kept the data stream simple. In practice, segment-at-a-time is applicable to larger tuples with more attribute-value fields.

4.5 Experimental Results

In this section, we evaluate our approach. We implemented the proposed design in VHDL and configured various OP-Chain topologies on the Xilinx ML505 development board to demonstrate the applicability of OP-Blocks as building blocks to develop custom stream processors. The detailed specification for the Virtex-5 FPGA on this board is given in Table 4.5.3.

Table 4.5.2: OP-Block resource utilization.

Component	FLOP/LATCH	LUT	MUXFX	CARRY	BRAM	DRAM	OTHERS
Buffer Manager-R	37	214	2	15	-	-	2
Buffer Manager-S	55	100	-	15	-	-	2
Window Buffer-R	-	-	-	-	2	-	-
Window Buffer-S	-	-	-	-	2	-	-
Bypass Unit	16	22	-	-	-	11	1
Coordinator Unit	4	19	1	-	-	-	-
Processing Unit	327	533	7	119	-	-	2

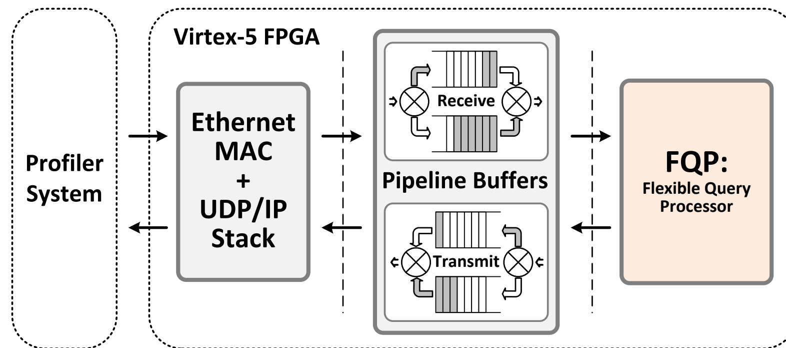


Figure 4.5.1: Data stream reception and transmission stages.

In our experiments, input is generated by a workload generator and passed through an Ethernet component and pipelined reception buffers to the stream processor, as shown in Figure 4.5.1. The input streams consist of 64-bit tuples (i.e., 32-bit attribute and 32-bit value).

We synthesized the OP-Chain design on our development workstation, loading the resulting bit file onto the FPGA using the JTAG interface. This bit file contains all required information to configure the FPGA. Our evaluation machine has an Intel Core i7-3720QM processor and 16 GB DDR3 RAM running Window 7 Professional (Service Pack 1). In each experiment, we used different parameters for a specific evaluation objective; parameters (specified by figure number) are given in Table 4.5.1.

R-Pipe, T-Pipe, Filter-U, and Buffer-N specify buffer sizes (in number of tuples) for reception and transmission pipelines, *Filter Unit*, and *Buffer Node*, respectively. *Buffer Node* is a two-to-one merger component including memory, used in each of the internal nodes of the *Result Aggregation Buffer*.[3] Finally, Count specifies the number of OP-Blocks in the experiment and Win-Buf specifies the size of each window buffer in an OP-Block.

[3] *Buffer Nodes* are connected in a binary tree-like structure to pass on results. Leaves are connected to OP-Blocks (i.e., to the *Filter Unit*) and the root is connected to the output port of the FQP.

Table 4.5.3: Virtex-5 (XC5VLX50T) specification.

Slices	8,160
Distributed RAM (Kb)	780
DSP48E Slices	288
BRAM Blocks (Kb)	4,752
RocketIO Transceivers	12
Max User I/O	480

Table 4.5.4: Instruction latencies in OP-Block (operating clock frequency: 125MHz).

Operation	Latency (#cycles)	Duration (ns)
Selection	6	48
Projection	5	40
Join	5	40
Reset	4	32
Bypass	2+2	32

4.5.1 OP-Block Analysis

The resource utilization, reported by the Xilinx synthesis tool, for an OP-Block capable of processing tuples with a 64-bit width (32-bit attribute, 32-bit value) and window buffers, each with the size of 64 tuples, and a buffer size of 4 tuples for the *Bypass Unit* (BU) is presented in Table 4.5.2. The data shows that the window buffers consume two block RAMs (BRAMs) and the buffer for the BU consumes 11 distributed RAMs (DRAMs).

LUTs realize logical operations and latches (flip flops) store values between operations. An OP-Block's PU consumes 533 and 327 of these resources, respectively, thus, accounting for more than 60% and 74% of them, respectively. This underlines the significant increase of relative chip area dedicated to raw processing power in our design as compared to the 5% area dedicated to the Arithmetic Logical Unit by general purposes processors. Also, the increase in size of the PU (i.e., adding more complex arithmetic operations or using multiple parallel execution units) has almost no effect on the size of other components, which means that the performance/transistor ratio (the raw processing power) has even more room for improvements in our design.

4.5.2 Query Insertion and Modification Latencies

Each instruction (i.e., query operator) takes a number of cycles to program the OP-Block based on factors such as the chosen topology and current system load. In case there are no stalls, latencies for instruction insertion are given in Table 4.5.4.

The latencies presented in Table 4.5.4 assume that all instruction chunks are available at the Stream S input port of the OP-Block. In case an instruction is not targeted at the current OP-Block, a bypass is performed in two steps. First, taking two clock cycles, the current OP-Block decides on the bypass based on the data available on its port (from its right neighbor) and starts the reception of the data. Second, also taking two cycles, the current OP-Block initiates a transmission to its left neighbor. Reception and transmission operate independently from one another by using the shared buffer in the BU until all instruction chunks have been bypassed.

The instruction insertion latencies demonstrate that query programming is accomplished in a matter

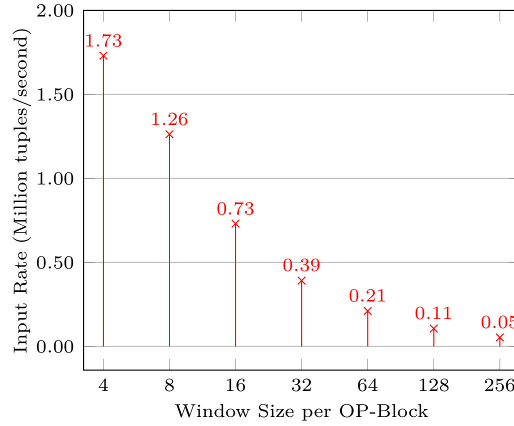


Figure 4.5.2: Effect of window size on input throughput.

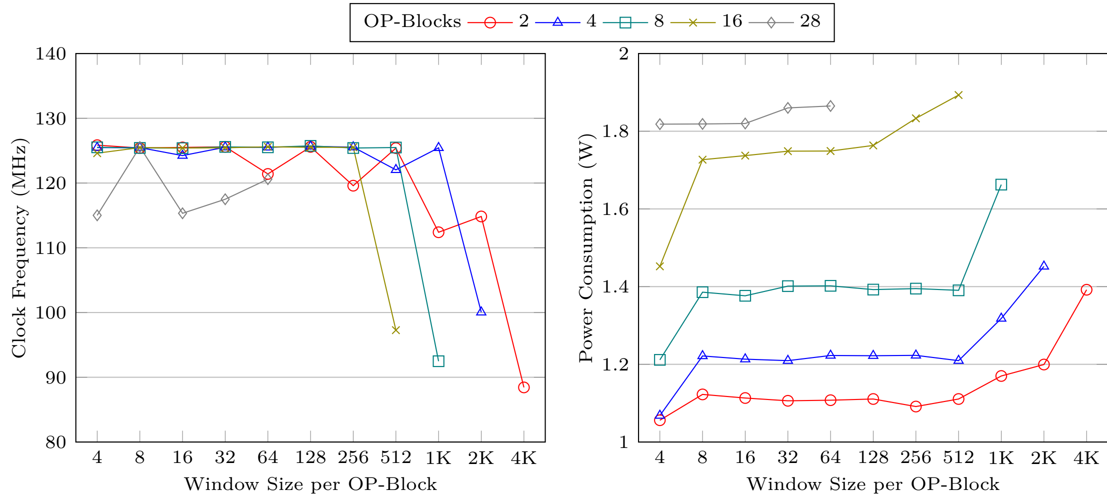


Figure 4.5.3: Maximum clock frequency of an OP-Chain on Virtex-5.

Figure 4.5.4: Power consumption report.

of nanoseconds, clearly, a viable alternative to the comparatively long synthesis times that range from hours to days and FPGA re-configuration times that range from seconds to minutes.

4.5.3 Performance Evaluation

The rate at which the processor can accept input, the *input throughput*, can be determined based on the number of OP-Blocks executing in parallel in addition to each OP-Block's window size. Figure 4.5.2 shows the maximum input rate sustained by an OP-Chain consisting of 16 OP-Blocks with window sizes varying from 4 to 256 tuples (for each OP-Block) running at a clock frequency of 125MHz. This chain realizes an equi-join operator on two input data streams. The number of processed tuples per second decreases as we allocate larger windows to each OP-Block, which is expected as each OP-Block requires more clock cycles to process a larger window.

To run controlled experiments, we define the *match probability* (MP) as the probability for two tuples to have the same value, i.e., resulting in a match for an equi-join.

Figure 4.5.7 shows the effect of MP on the maximally sustained input rate (red diagrams) and maximally sustained output rate (blue diagrams). These experiments are based on an OP-Chain instance with 16 OP-Blocks in two configurations of 8 and 16 tuples per window in each OP-Block.

In the OP-chain configuration, the maximum input rate of $\sim$1.26 million tuples per second can be sustained by the available output bandwidth. However, an increase in MP, impacts the tuples produced as output, saturating the output bandwidth that then again aversely effects the input rate. At an MP of 0.05 (5%), the output bandwidth is saturated, which decreases the output rate: More tuples match as part of a join and are forwarded as results.

4.5.4 Scalability Evaluation

Figure 4.5.3 shows reported clock frequencies for various OP-Chain configurations with different number of OP-Blocks and window buffers. The results show close to constant clock speeds as the design gets bigger. In hardware design, this is an indication for the scalability behavior of a design, i.e., in our case, with increasing number of OP-Blocks, the clock frequency at which the FQP runs remains unchanged.

For window sizes beyond a certain limit, the clock frequency an OP-Chain can support significantly drops due to the design footprint approaching the given FPGA's resource limits. For the FPGA we have available, this materializes as follows: For 2 blocks and a 2K window size and for 4 blocks and a 1K window size (8 and 512; 16 and 256.)

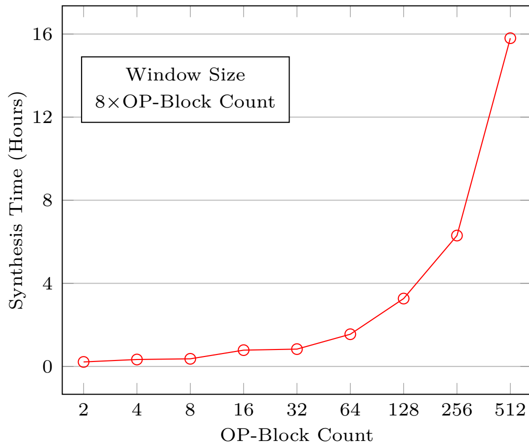


Figure 4.5.5: Synthesis times on Virtex-7 (XC7VX980T).

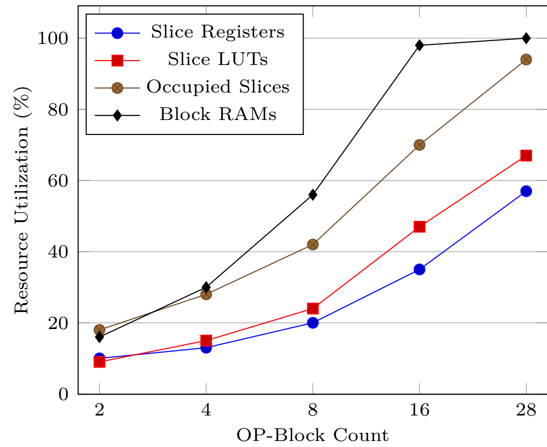


Figure 4.5.6: OP-Chain resource utilization on Virtex-5 (XC5VLX50T).

The synthesis tool aims to utilize every available FPGA resource to realize a given design. However, when a design becomes larger, it has fewer options available (e.g., for placement and routing). As a result, the synthesis tool may end up mapping components far from each other, which degrades the sustainable clock frequency for the realized design. This limitation is known by FPGA designers, who generally recommend a 70-80% resource utilization for the given FPGA to prevent a clock frequency drop [29]. There are optimization options in synthesis tools to mitigate this issue like options aiming at trading off speed for area and options to tune the amount of optimization to apply during synthesis to mapping, placement, and routing. Playing with these parameters could result in better configurations. However, since logic synthesis is time-consuming, studying the detailed effects of each parameter is beyond our current means.

Figure 4.5.6 shows the percentage of utilized resources on the Virtex-5 (XC5VLX50T) FPGA. Here, the window size is fixed to 8 tuples for each OP-Block.

Synthesis tools commonly favor speed (i.e., higher clock frequency) for smaller designs. As a result, FPGA resources (i.e., slices and BRAMs) may be utilized partially. This effect is seen in the BRAM utilization between OP-Chains with 16 and 28 OP-Blocks. Figure 4.5.6 shows that for the former OP-Chain almost all BRAMs are allocated, while we were still able to realize the latter and bigger OP-Chain.

Resource utilization information is beneficial to determine underutilized FPGA resources to further improve design performance by tuning.

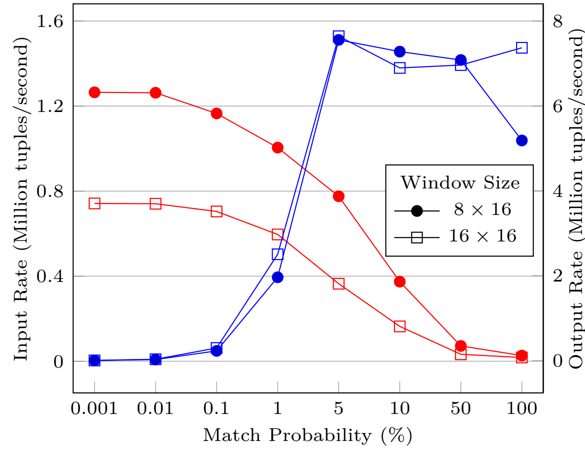


Figure 4.5.7: Effect of match probability on input (red) and output (blue) throughput.

4.5.5 Power Consideration

Power consumption is an important design requirement that impacts the energy budget alloted to a given component and the rate of thermal dissipation, which is a concern for chips as their packaging (die) size shrinks.

Figure 4.5.4 reports on power consumption estimates of various OP-Chain configurations based on default activity rates reported by the ISE Xilinx XPower Analyzer (Release: 14.4 - P.49d). Static power consumption remains almost constant (∼670 milliwatt), which is an indication that all resources are powered regardless of their utilization in this FPGA.

Our measurements show a direct relation between power consumption and number of OP-Blocks. For lower OP-Block counts, static power consumption dominates, while for larger counts, dynamic power consumption dominates the total chip power consumption. Consumption ranges from 1 to 2 watts, which is comparably low given the ∼100 watts power draw of modern CPUs.

4.5.6 Synthesis Challenges

Here, we measure the synthesis time for various different configurations of our stream processor. Figure 4.5.5 reports synthesis times for OP-Chain designs with varying number of OP-Blocks.

Our synthesis time measurements do not include the Ethernet MAC module and the pipeline reception and transmission buffers (cf. Figure 4.5.1). Additionally, synthesis optimization options were kept to default settings.

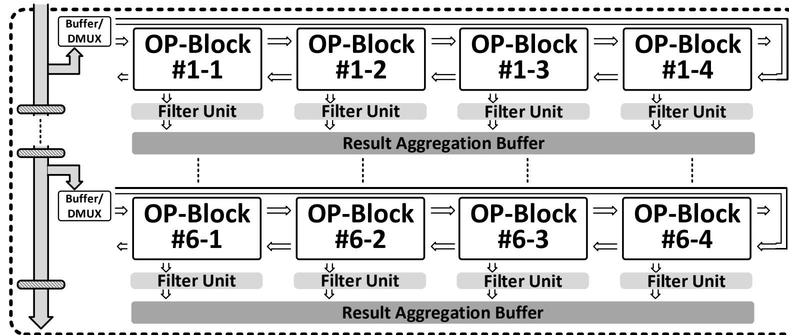


Figure 4.5.8: Partially parallel FQP topology.

Table 4.5.5: Tuple processing rate.

Operators	# Operators	Million Tuples/s
Selection	24×8	230.6
Projection	24×16	272.6
Join	24	34.5
Chained Join (4)	6	8.6

The measurements show that synthesis times grow rapidly as the design includes more OP-Blocks. Unfortunately, today's synthesis tools do not seem to be capable yet to fully exploit the parallelism available in multi-core processors in the synthesis computations. This is mostly due to the complexity inherent in the various synthesis process steps (i.e., heuristic nature due to NP-hardness of underlying synthesis problems).

From a high-level, these measurements underline again the main advantage of our design, as being the capability to flexibly process queries at run-time without having to incur expensive re-synthesis overhead.

4.5.7 Segment-at-a-time Processing

We developed all FQP components in VHDL that are configured and synthesized on our Xilinx ML505 development board. In our experiments, the input was generated by a workload generator and passed through an Ethernet component and pipelined reception buffers to the FQP stream processor. The input streams consist of 64-bit long tuples (i.e., 32-bit attribute and 32-bit value).

Raw Processing Power Evaluation —

We first present the raw processing power of various queries by focusing on the number of operators on a topology similar to Figure 4.5.8, where window size is 16 and clock frequency is 125MHz. For the selection and projection operators, OP-Block is capable of supporting |Window Buffer-L|/2 independent selection operators or |Window Buffer-L| independent projection operators. Each

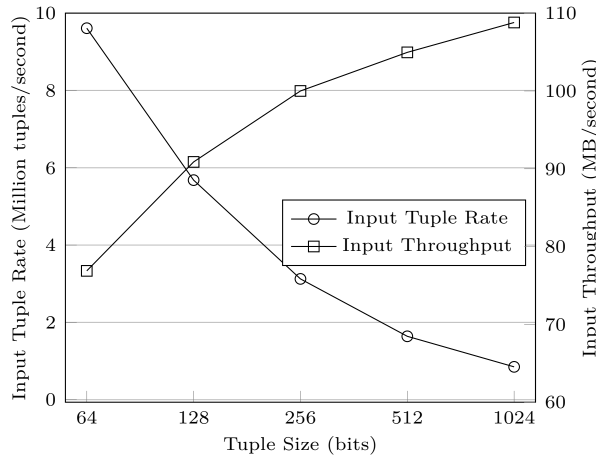


Figure 4.5.9: Effect of tuple size on input tuple rate and input throughput.

OP-Block is capable of realizing a single join operator. OP-Blocks connected in a chain (OP-Chain) can realize join operators with even larger window buffers. For example, utilizing two, three, or four OP-Blocks increases the window size two, three, or four times, respectively. The processing performance of each OP-Block for the join operator tightly depends on its window buffers' sizes. For a window size of 16 tuples, the current version of OP-Block is capable of processing 1.44 million tuples per second. The raw processing power of the topology given in Figure 4.5.8 is summarized in Table 4.5.5.

Each OP-Block is capable of processing at the rate of 9.61M, 11.36M, or 1.44M tuples per second for the selection, projection, and join operator, respectively, which translates to 230.6M, 272.6M, or 34.5M tuples per second for the topology in Figure 4.5.8. By chaining 4 OP-Blocks we have 6 OP-Chains each with a window size of 4×16 and a total processing rate of 8.6M tuples per second.

Segment-at-a-Time Evaluation —

To evaluate our segment-at-a-time feature of OP-Block, and to study its influences on the input rate, we utilized a data stream by varying the number of attribute-value pairs per tuple (1 to 16), in which the size of each attribute-value pair is 64 bytes. The clock frequency in this experiment was 125MHz. Figure 4.5.9 demonstrates the input tuple rate achieved as we feed larger tuples to an OP-Block. By feeding larger tuples, the sustainable input tuple rate decreases as expected, since the size of tuple and the number of attribute-value pairs doubles each time. However, interestingly for a double size tuple the processing time does not necessarily double as seen in this figure. This is due to the reduction in the amortized cost of tuple handling that is mostly for the first segment and decreases for the subsequent segments. These results are for the selection operator, but they are applicable for other operators including the projection and join operators.

Chapter 5

Parallel Stream Join Architecture

There is a rising interest in accelerating stream processing through modern parallel hardware, yet it remains a challenge as how to exploit the available resources to achieve higher throughput without sacrificing latency due to the increased length of processing pipeline and communication path and the need for central coordination. To achieve these objectives, we introduce a novel top-down data flow model for stream join processing (arguably, one of the most resource-intensive operators in stream processing), called SplitJoin, that operates by splitting the join operation into independent storing and processing steps that gracefully scale with respect to the number of cores. Furthermore, SplitJoin eliminates the need for global coordination while preserving the order of input streams by re-thinking how streams are channeled into distributed join computation cores and maintaining the order of output streams by proposing a novel distributed punctuation technique. Throughout our experimental analysis, SplitJoin offered up to 60% improvement in throughput while reducing latency by up to 3.3X compared to state-of-the-art solutions.

This section presents the following contributions:

1. we propose SplitJoin, a novel scalable stream join architecture that is highly parallelizable and removes inter-core communications and dependencies,
2. we introduce a new splitting abstraction in SplitJoin to "process" and "store" incoming data streams concurrently and independently,
3. we propose a top-down data flow model to achieve a coordination-free protocol for distributing and parallelizing stream join processing,
4. we develop a distribution tree with logarithmic access-latency for routing of incoming data to storage and processing cores, while preserving the ordering of incoming tuples,

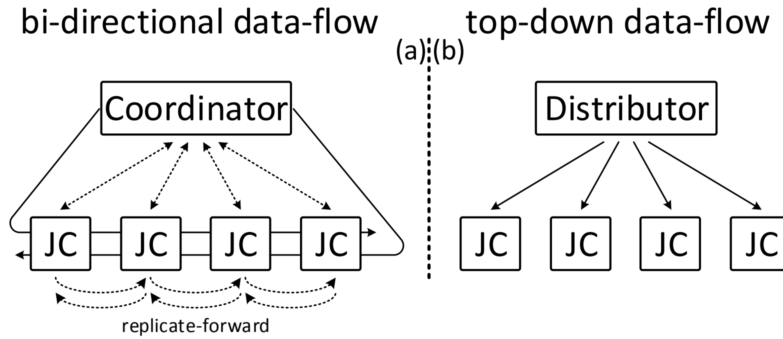


Figure 5.1.1: Stream join data flow models (*JC* stands for join core): (a) bi-directional and (b) uni-directional (top-down).

5. we design a coordination-free protocol that does not rely on global knowledge to produce ordered join output streams by proposing a relaxed adjustable punctuation (RAP) technique with tunable precision, and
6. we conduct an extensive analytical and experimental study of SplitJoin as compared to existing state-of-the-art solutions.
7. Present a case study for design decision importance for hardware acceleration, including design and development of a general flow-based framework to realize a parallel stream join based on two bi-flow and uni-flow models on hardware.
8. Explore uni-flow and bi-flow hardware design decisions and their effects on throughput, latency, and power consumption.

5.1 SplitJoin

In this section, we describe SplitJoin and highlight two of its key properties, namely, the top-down data flow and the splitting of the join computation into independent storage and processing steps. Together, these properties remove any need for coordination and dependencies among join cores, which enables a high-degree of parallelism for SplitJoin without sacrificing latency.

5.1.1 Overview

SplitJoin diverts from the bi-directional data flow-oriented processing of existing approaches [85, 75]. As illustrated in Figure 5.1.1, SplitJoin introduces a single top-down data flow that fundamentally changes the overall tuple processing architecture. First, the join cores are no longer chained linearly (i.e., avoiding linear latency overhead). In fact, they are now completely independent (i.e., also

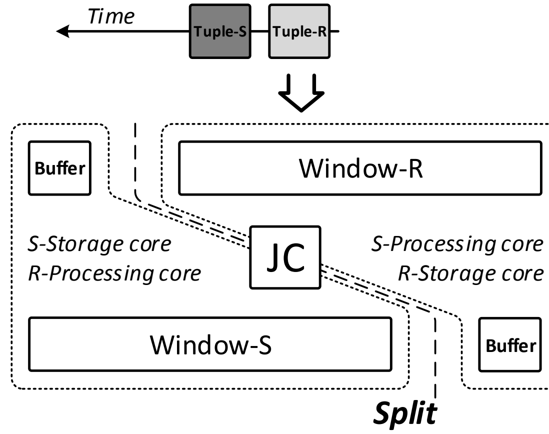


Figure 5.1.2: SplitJoin concept.

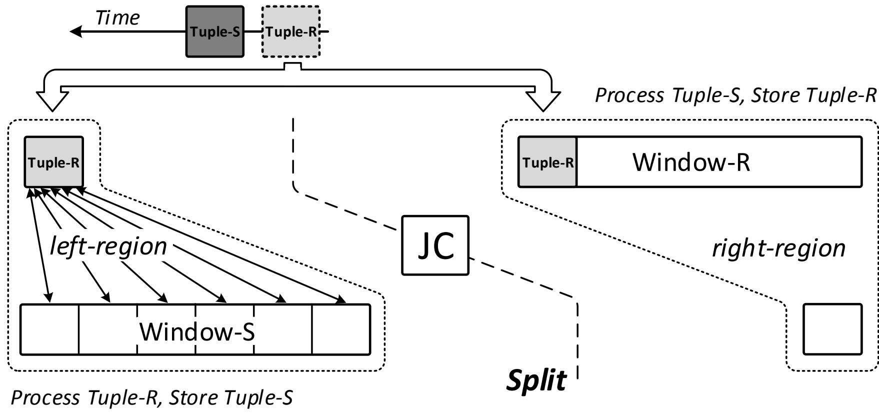


Figure 5.1.3: SplitJoin storing and processing steps.

avoiding inter-core communication overhead). Second, both streams travel through a single path entering each join core; thus, eliminating all complexity due to potential race conditions caused by in-flight tuples and complexity due to ensuring the correct tuple-arrival order, namely, the FIFO property is trivially satisfied by using a single (logical) path. Third, the communication path can be fully utilized to sustain the maximum throughput and each tuple no longer needs to pass every join core.

Another important aspect of SplitJoin is the simplification and decomposition of join processing itself. SplitJoin splits the dominant join abstraction that enforces the "storing" and "processing" steps to be coupled and done in a serial order. SplitJoin views these steps as two independent steps, namely, (i) "storing" and (ii) "processing". In fact, SplitJoin goes one step further and shows that not only these steps could be done in parallel, they can also be distributed to independent join cores. Therefore, unlike traditional parallel join processing that divides a single window into a set of sub-windows, where each is assigned to a core, SplitJoin introduces separate *storage* and *processing* cores that

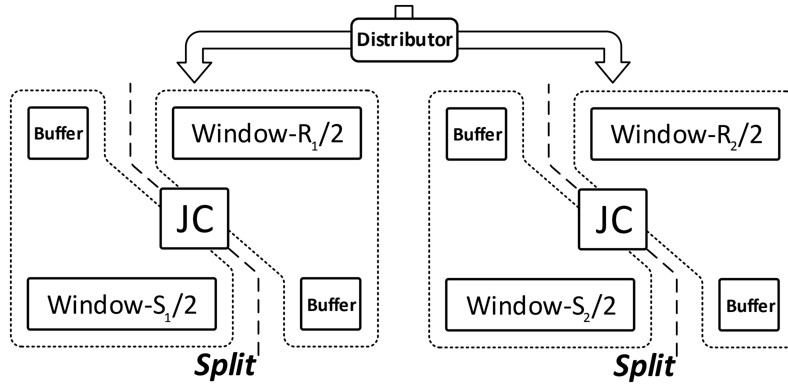


Figure 5.1.4: SplitJoin parallel architecture.

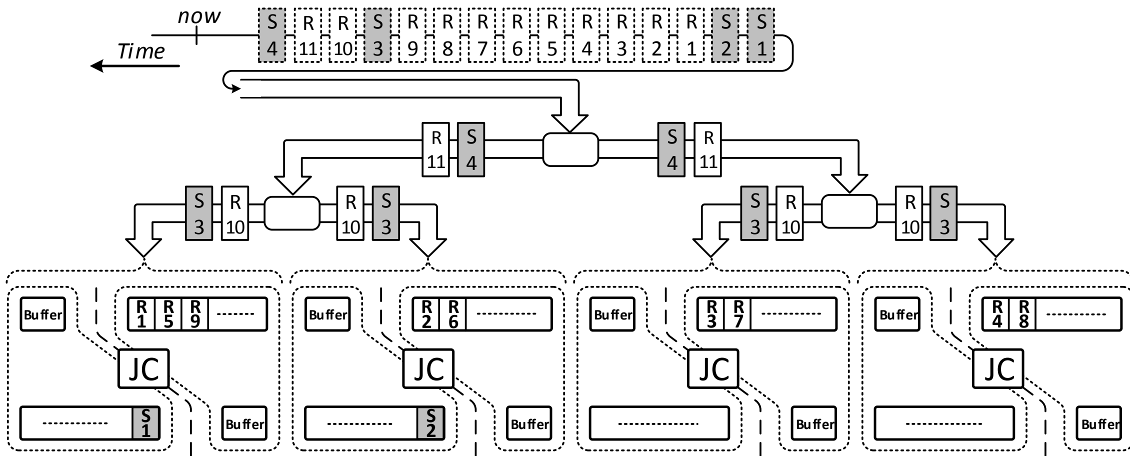


Figure 5.1.5: SplitJoin data distribution and processing example.

operate independently of each other as shown in Figure 5.1.2. The storage core is responsible for storing new tuples, while the processing core is responsible for the actual join operation of a new tuple in one stream with the existing tuples in the other stream. The splitting line in Figure 5.1.2 conceptually divides our join processing architecture into two separate parts, in which a *region* represents a stream's window and the associated buffer. We use the term *right-region* when referring to *Window-R* and *left-region* for *Window-S*. For each incoming tuple, a region either does processing or storing.

The split mechanism is illustrated in Figure 5.1.3, where the incoming tuples are fed to SplitJoin one after another. In the first step, `Tuple-R` is inserted into both regions. The right-region is responsible for storing `Tuple-R` in its sliding window, while the left-region is responsible for the processing of the replicated copy of `Tuple-R` (i.e., the join comparison). The temporary tuple replication eliminates all inter-region communication among storage and processing cores. The replicated tuples are simply discarded once the processing is completed.

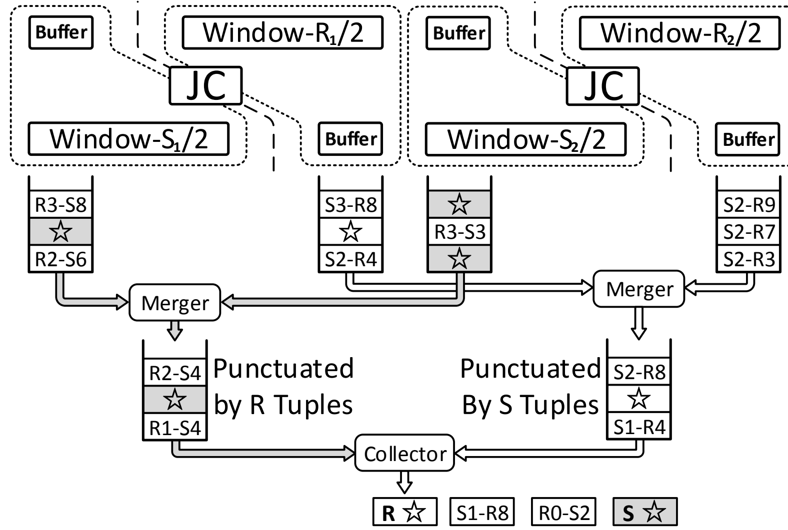


Figure 5.1.6: Punctuation in result gathering.

5.1.2 SplitJoin Parallelism

In SplitJoin, we parallelize the stream join computation by dividing each sliding window into a set of disjoint sub-windows. Each sub-window is assigned independently to a join core as shown in Figure 5.1.4 (i.e., acting as a local buffer for each core). Each join core (*JC*) consists of a *left-* and a *right-region.* The division of the sliding window among join cores is accompanied by a *Distributor* unit to transmit incoming tuples to the join cores.

In the parallelized version of SplitJoin, all join cores receive the new incoming tuple. In each join core, depending on the tuple origin *i.e.*, whether R or S stream, the processing and storage steps are orchestrated. For example, if the incoming tuple belongs to the R stream, `Tuple-R`, then all processing cores dedicated to the left-region compare `Tuple-R` against all the tuples in the S stream sub-windows. Simultaneously, `Tuple-R` is also stored in the storage core of exactly one right-region. The assignment of `Tuple-R` follows an arbitration of the tuple to a storage core based on a round-robin selection. In other words, each region, based on its position number [1] and the number of seen tuples, independently determines its turn to store an incoming tuple. The proposed assignment model eliminates the need for a central coordinator for tuple assignment, which is a key contributor for achieving scalability in SplitJoin architecture. Notably, transmitting an incoming tuple to each join core translates into writing a tuple to the join core's local buffer (independent of any other join cores) that resembles a simple queue with a single producer and a single consumer, in which the producer is the *Distributor* and the consumer is join core itself.

[1]Position number refers to the logical location of a join core among other join cores.

5.1.3 Scalable Distribution Tree

The decoupling of storage and processing in SplitJoin simplifies parallelization by distributing sub-windows among many independent join cores. To fully leverage potential parallelism, we also need an efficient tuple distribution and routing mechanism.

In SplitJoin, to distribute the stream's transmission load in a balanced and scalable manner, we use a *k-ary* tree as the distribution network. As the network grows in size, the *Distributor* is replicated and its replicas are placed in the tree's inner nodes to achieve the desired scalability. As the number of SplitJoin join cores increases, we increase the fanout of each *Distributor* before increasing the depth of the distribution tree.

By applying replication recursively, we scale the distribution network as well as the number of join cores for SplitJoin. The resulting system, including the input data distribution network, SplitJoin's join cores, and the output data gathering network (similar in structure to the input network), is shown in Figure 5.3.2, where the horizontal bars illustrate the input distribution and output gathering networks.

The distribution network is the same for both count-based and time-based sliding window joins. However, in the time-based version each tuple carries an extra field for its timestamp. This field is to keep track of the lifespan of each tuple to realize the time-based sliding window semantic.

5.1.4 Expiration and Replacement Policies

Tuple expiration is a crucial step to ensure the correctness of the stream join semantic. In the count-based sliding window, the number of tuples in each window is specified explicitly while in the time-based sliding window a lifespan l (e.g., $l = 10$ minutes) defines when a tuple must be expired.

SplitJoin supports both passive and active expiration techniques. The passive approach is primarily intended for the count-based sliding window, in which the incoming tuples simply overwrite the oldest tuples in the window. The expiration is done implicitly and mimics the functionality of a FIFO buffer. Once a window is full, the stored tuples are expired in order of their arrival. In the active expiration, geared towards the time-based sliding window, each join core locally manages the expiration of tuples from its sub-window. The expiration task for each sub-window is postponed until a tuple from the opposing stream with timestamp of t is received for processing. Then, in the region responsible for processing, just prior to the join computation, for each tuple with a timestamp t_i, if $(t - t_i) > l$, then the tuple is expired. Basically, tuples are expired when they fall off the user-defined lifespan (l) of the time-based window size.

Note the expiration is a local operation within a region and does not involve global coordination because tuples arrive with monotonically increasing timestamps and order is preserved when they are added and stored in a sub-window. The expiration task starts from the end (with the oldest tuple) of each sub-window and ends when a tuple younger than the user-defined lifespan is found. In other words, instead of sending explicit expiry messages with a timestamp, we rely on the timestamp of tuples in the input streams that must be routed to all nodes anyway. Therefore, the expiration messages are implicitly piggybacked on the incoming tuples as a way to broadcast the synchronized time without the need for global coordination.

In Figure 5.1.5, we illustrate how tuples are stored and processed in SplitJoin join cores. Assuming that we have a sequence of tuples as shown in the upper part of the figure, each tuple is transferred by the distribution tree to all join cores. Each `Tuple-R` is stored in exactly one *right-region* while processed by the *left-region* of all join cores. Likewise, for tuples from the S stream, they are stored in the *left-regions* and are processed by the *right-regions*.

As we can see in Figure 5.1.5, the tuples are distributed in-order. The tuples reach the storage cores through the same path (i.e., the top-down flow), and the expiration procedure is preformed based on the order of incoming tuples (for both count-based and time-based sliding windows). Thus, unlike the bi-directional model used in [85, 75], neither the concurrency nor the race condition issues arise.

During processing, each region emits resulting tuples to be collected by the result gathering network. The processing step for each tuple in each region is completed by emitting an end notice from that region, referred to as a *star* punctuation mark. These marks serve to preserve the order of join results as we describe in detail in Section 5.2.

5.1.5 SplitJoin Algorithms

Tuple distribution in SplitJoin is specified in Algorithm 1. Upon arrival of a new tuple, regardless of its source stream, the tuple is broadcast to all join cores (Line 3).

Algorithm 1: SplitJoin distribution network.

```
1 SplitJoin() begin
2 |  while still a tuple to consume do
3 |  |  broadcast tuple t to all Join Cores begin
4 |  |  |  forall join cores do
5 |  |  |  |  Join_Core(t, source);
```

In each join core, as presented in Algorithm 2, depending on the tuple's source (Lines 2 and 11), from R or S stream, the tuple is sent to the *right-region* for storage and to the *left-region* for processing or vice versa.

Algorithm 2: A join core in SplitJoin.

```
1  Join_Core(t, source) begin
2      if source = Stream R then                                   // right-region
3          Expiration_Process(t, sub-window S);
4          Processing_Core(t, sub-window S);
5          if R_store_counter = node_id then
6              Storage_Core(t, sub-window R);
7          if R_store_counter = number of join cores then
8              R_store_counter ← 0;
9          else
10             R_store_counter ← R_store_counter + 1;
11     else                                                        // left-region
```

Algorithm 3: Matches between **t** and sub-window **X**.

```
1  Processing_Core(t, sub-window X) begin
2      forall t_i-tuple in sub-window X do
3          compare t_i-tuple with t; if match then
4              emit the matched result;
5          if i ≡ 0 (mod ordering_precision) then
6              emit punctuation star;
```

Finally, in the expiration process specified in Algorithm 4, the tuples that are too old to be considered for the join are expired from the end of the sub-window by computing their lifespan using their timestamp and the timestamp of the new tuple.

The pseudo code for the processing core (i.e., the join comparison) is specified in Algorithm 3. An incoming tuple is compared with all tuples in the opposite sub-window (Lines 2-4). More importantly, this step is executed concurrently for each sub-window in every region. After processing (Lines 5-6), based on the chosen ordering precision, the *star* marker is produced and emitted. Also note that the goal of SplitJoin is to provide an efficient and coordination-free architecture for performing stream joins, and the particular choice of join algorithm is orthogonal. In this work, we adopted a simple variation of nested-loop join; however, within each core, one may choose any join algorithms such as hash- or index-based join.

Algorithm 4: Expiring old tuples for time-based version.

```
1  Expiration_Process(t, sub-window X) begin
2      i ← the end of sub-window X;
3      while t_i.timestamp - t.timestamp > Time Window Size do
4          omit t_i from sub-window X;
5          i ← i − 1;
```

5.2 Punctuated Result Collection

In SplitJoin, we employ a result gathering network (similar to our data distribution network) and a punctuation technique to preserve the ordering for the join result output. The full architecture of SplitJoin, that includes the distribution network, join cores (*JCs*), and the collection network, is illustrated in Figure 5.3.2.

In SplitJoin, we utilized a *2-ary* collection tree to gather and merge join results as depicted in Figure 5.1.6. The result tuples of each processing core are gathered from the leaves of the collection tree. Each core has its own dedicated FIFO buffer. The collection tree employs a *Merger* unit and a FIFO buffer in each of its intermediate nodes (except in the root). Moving toward the tree's root (from top to bottom), at each node, the data in the two input buffers is merged into the buffer of that node. Merging continues up to the root, which contains the last buffer emitting the gathered join results.

5.2.1 Punctuation-based Ordering

SplitJoin architecture preserves the ordering of result tuples. The precision of the output order can be determined by a tunable system parameter, without significant changes in the processing architecture. To realize this flexibility in our design, we developed a *relaxed adjustable punctuation* (RAP) strategy. We define two levels of ordering guarantees for join results: the *outer* and *inner ordering*.

Definition 1. The *outer ordering* of join results ensures that for any two consecutive incoming tuples, join results of the first tuple always precede the join results of the second tuple.

Definition 2. The *inner ordering* of join results ensures that for a single incoming tuple in one stream, join results are ordered in ascending order from the oldest to the most recently inserted tuple in the other stream.

Our proposed relaxation enables us to maintain strict outer ordering while adjusting the precision of the inner ordering (essentially, not maintaining the inner ordering) in order to substantially reduce the overall cost of ordering. Furthermore, our technique supports strict outer and inner ordering as well.

In RAP, we define a simple punctuation emission rule for each core (the same simple rule applies to all cores), that is, the emission of a punctuation at the end of the processing of every newly inserted tuple (preserving the outer ordering and relaxing the inner ordering). In other words, each join core emits a punctuation after the end of processing a newly inserted tuple with all tuples in the other window. We differentiate this punctuation from result tuples by a *star*, as shown in Figure 5.1.6.

SplitJoin cores insert both the join results and punctuation marks to collection tree leaves. The punctuation acts as a border between the join results of two consecutively inserted tuples (outer order). As join results and punctuations are pushed down the collection tree towards the root, at each node of the tree, the join results and their corresponding punctuation marker (stars) from the two buffers are merged into the FIFO buffer of their parent node. When the *Merger* in the parent node receives a star from one of its inputs, it disables that input and continues to receive resulting tuples from the other buffer until it receives a star from that buffer as well. The *Merger* merges

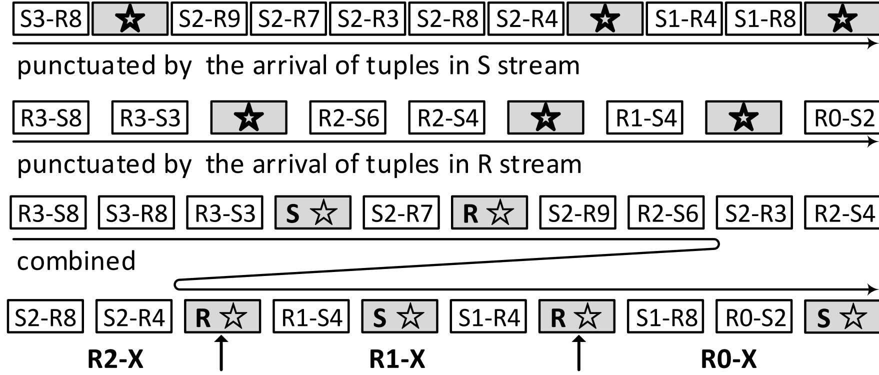


Figure 5.2.1: Punctuated resulting stream.

two punctuations (stars) into one and pushes it to its FIFO buffer. This scenario repeats until the star reaches the output of the collection tree.

Since join results are pushed down in the order in which the newly inserted tuple arrives, the outer ordering for each core is trivially satisfied due to the single top-down FIFO flow of SplitJoin that starts from the root of the distribution tree (for inserting new tuples) and ends at the root of the collection tree (for merging the join results). This flow is shown in Figure 5.2.1.

The final step in the result gathering network employs a *Combiner* rather than a *Merger*. On the right side of the split are the punctuated results, ordered by the tuples from the S stream, while on the left side, the punctuation is based on the arrival sequences of tuples from the R stream. These two sets of punctuated result tuples are consumable as separate streams. However, to emit only one stream as output, we use a *Combiner* which simply fetches the resulting tuples and punctuations from their input and puts them into the output buffer. The *Combiner* keeps track of the origin of punctuations (whether from the right- or left-regions) by flagging the *stars* with R and S, as shown in Figure 5.2.1.

In Figure 5.2.1, the upper flow is the result stream from the right-regions, punctuated by the order of S stream tuples, while the middle one is the result stream from the left-regions, punctuated by the order of R stream tuples. The lower flow demonstrates the combined result stream that includes all result tuples in addition to punctuations. For example, **R1-X** is specified by two R punctuations and includes the result tuples which start with **R1**.

Adjusting the punctuation interval is straightforward and only requires us to tune the punctuation emission rate in SplitJoin's cores. Each core can simply change the frequency at which a punctuation is generated. For example, each core can be tuned to produce a punctuation after joining one newly inserted tuple (strict outer ordering) or after every five tuples (relaxed outer ordering). We could also adjust the precision of inner ordering by increasing the frequency of punctuation generation.

Algorithm 5: Punctuation-based N-ary merger.

```
1 N-ary_Merger(t) begin
2     foreach right(or left)-region of core1..N in sequence do
3         while a resulting tuple (t) is available in output buffer till the first star do
4             pop t from join core's output buffer;
5             push t to Merger's output buffer;
6     push out the end of result star;
```

For example, to produce a strict inner ordering, each incoming tuple is compared with tuples in the opposite window (starting from the oldest to the most recently inserted one), followed by outputs for both the join result and the punctuation marker for every comparison. Therefore, if each core has a window size of w, then up to w punctuation markers (i.e., *stars*) are produced for every newly inserted tuple. For a relaxed inner ordering, only one punctuation is produced after joining the incoming tuples with all the tuples in the opposite window. At the other extreme, when no ordering is required, we could simply disable the punctuation generation altogether.

5.2.2 Ordering Algorithm

For the result gathering network, we utilized a *k-ary* tree. Algorithm 5 specifies the pseudo-code for an *N-ary* (e.g., 2-ary) *Merger* given an *N-ary* result gathering tree. The *Merger* is connected to the output FIFO buffer of N regions and collects the resulting tuples and punctuations into its own output FIFO buffer, which is subsequently fed to the next intermediate node (its parent) in the tree. This is repeated up to the root of the tree, where result tuples are punctuated by tuple arrival order from the two stream types (either R or S).

The *Merger* connects to the output buffers of the same source, either *left* or *right* regions (cf. Line 2 of Algorithm 5). Each *Merger* collects the results in the same order as the join cores store the new incoming tuples. For example, assuming that the first `Tuple-R` is stored in the left most join core in its *right-region*, as shown in Figure 5.1.5. The *Merger* then begins the collection of results from the comparison of `Tuple-S` with the R sub-window in the left most *right-region* as well.

In the result gathering, the *Merger* fetches tuples from the first region's buffer and stores them in its own output buffer until it reaches the first *star* in (Line 3∼Line 5). Then it repeats the same procedure for the next region's buffer until it receives a *star* from there too.

After receiving a punctuation mark from the last region, the *Merger* forwards the punctuation to its output buffer (cf. Line 6). Note that each *Merger* emits only one punctuation mark for every pair of punctuation (i.e., one punctuation mark from each join core).

Using a higher ordering precision increases the number of punctuation marks between result tuples of each region. For example, instead of having one punctuation mark after comparison of a tuple

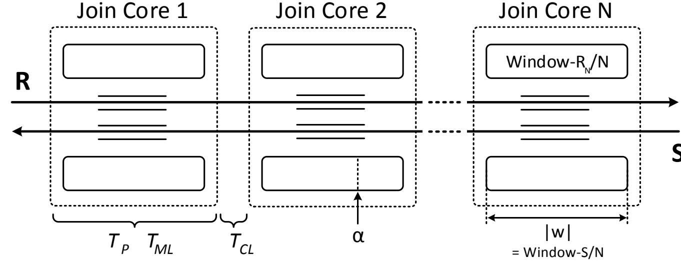


Figure 5.3.1: Low-latency handshake join overview [75].

with the whole sub-window, we can have one punctuation after each 10 comparisons. Since tuples in the sub-window are already stored in the order of their arrival, the intermediate punctuations preserve the result ordering while gathering the results from all the join cores. For obtaining a higher precisions, *Mergers* follow the same procedure as before.

5.3 Runtime Complexity

In this section, we present a brief analytical model to study the runtime complexity of SplitJoin relative to related techniques [85, 75]. In the analysis, we use the following definitions.

Definition 3. We define the *processing latency (PL)* as the time from when a tuple arrives at the join operator until the tuple is compared and joined with all tuples in the other window and all the matching results are produced.

Definition 4. We define the *visiting latency (VL)* as the time required for two tuples from both streams to be compared with each other.

5.3.1 Low-latency Handshake Join Analysis

The processing latency for low-latency handshake join (cf. Figure 5.3.1) is given as follows:

$$PL = T_{CL} + ((k-1) \times (T_{CL} + T_{ML})) + (w \times T_P) + T_{Col} \tag{5.3.1}$$

where T_{CL} represents the communication time between cores and k the number of processing cores. T_{ML} accounts for the tuple monitoring time in both streams, required to prevent missing results between tuples in the fast-forwarding buffers (i.e., race conditions). Therefore, the cost of propagating a single tuple to all cores by replication and fast-forwarding is captured by $((k-1) \times (T_{CL} + T_{ML}))$. The size of each sub-window in each core is denoted by w. T_P represents the processing time to perform the join operation between each pair of tuples. To simplify the analysis, we assume that

all join cores are working in parallel. Finally, T_{Col} presents the time required to collect all the matching results. In [75] the authors rely on a linear collector method for gathering the results that has the potential to break the strict neighbor-to-neighbor communication model of handshake join.

In theory, assuming a fixed-size sub-window, we can increase the number of processing cores to support larger windows. Therefore, the join's latency scales linearly in the number of processing cores, i.e., as $O(k)$, — optimistically assuming that central coordination would not become a bottleneck while ignoring the effect of the result collection method.

To calculate the visiting latency, we assume that the number of in-flight tuples from the two streams that must be compared (i.e., the monitoring time T_{ML}) is negligible. While this assumption renders the model less realistic (which was also implicitly assumed in [75]), it simplifies the visiting latency analysis.

Any pair of tuples from both streams meet each other in, at most, one location; let this location be α as shown in Figure 5.3.1. α could be in any core. If α happens to be on the first core, then the latency is lower, while if it is on the last core, then the latency is higher. Thus, we define the average visiting latency as follows:

$$VL_{avg} = T_{CL} + (\lfloor \frac{(k-1)}{2} \rfloor \times (T_{CL} + T_{ML})) + ((\frac{w}{2}) \times T_P) \tag{5.3.2}$$

$(\lfloor \frac{(k-1)}{2} \rfloor \times (T_{CL} + T_{ML}))$ determines the average time to reach location α (essentially, reaching the mid-point of core chain) and $(\frac{w}{2}) \times T_P$ captures the processing time for half the tuples at α. The visiting latency scales linearly with the number of processing cores $O(k)$, assuming $(T_{CL} + T_{ML})$ is constant, irrespective of the number of cores.

To simplify the analysis, we ignore the overhead of central coordination in low-latency handshake join [75]. The coordinator requires sending an explicit expiry message for every tuple [75]. On average, these messages double the communication traffic between the central coordinator and each join core, significantly affecting the performance as observed in our experimental evaluation.

5.3.2 SplitJoin Analysis

SplitJoin utilizes a distribution tree to deliver incoming tuples to each join core in $O(log_b k)$ time, where k is the number of join cores and b is the branching factor of the distribution tree. We define $Path_{1 \cdots k}$ as a distribution route that a tuple must travel to reach the join cores $1 \cdots k$, respectively. Let $T_{C_{Path_i, Depth_j}}$ be the communication cost (duration) of transferring a tuple to the ith path at

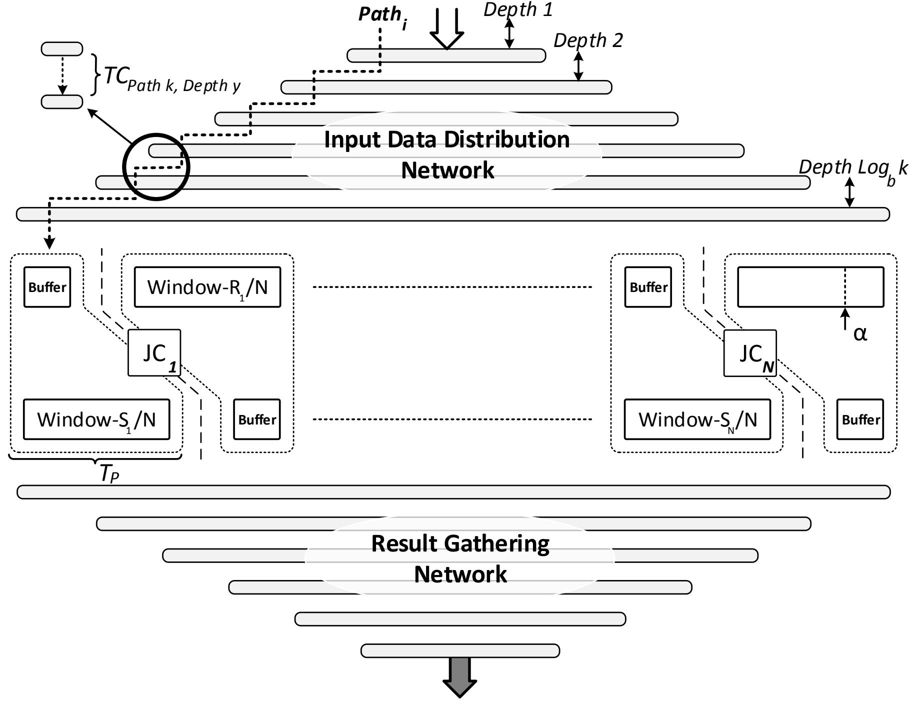


Figure 5.3.2: SplitJoin complete system.

depth j. We define the processing latency for SplitJoin as follows:

$$PL = \max_{i=1\cdots k}(\sum_{j=1\cdots log_b k} T_{C_{Path_i,Depth_j}} + (w \times T_{P_i})) + T_{Col} \tag{5.3.3}$$

where T_{P_i} is the processing time to perform the join operation between each pair of tuples for the i_{th} core. Assuming the communication times, $T_{C_{Path_i,Depth_j}}$, are roughly equal, then it follows that:

$$PL = \max_{i=1\cdots k}(T_{C_{Path_i}} \times log_b k) + (w \times T_{P_i})) + T_{Col} \tag{5.3.4}$$

If we further assume homogeneous join cores and homogeneous distribution routes within the tree and also decompose T_{Col} into smaller units of work, then it follows that:

$$PL = (T_{CL} \times log_b k) + (w \times T_P) + (T_{CL} \times log_c k) \tag{5.3.5}$$

where $\log_c k$ defines the depth of the result gathering tree with the branching factor of c (from root to leaves). Assuming a fixed-size sub-window, as we increase the number of join cores, latency increases logarithmically, $O(\log_b k)$ (assuming $b < c$), for SplitJoin as opposed to the linear increase $(O(k))$ observed in [75].

Supposing two consecutive tuples from both streams meet at the point α, as shown in Figure 5.3.2, then their communication times in the distribution tree mostly overlap with each other because

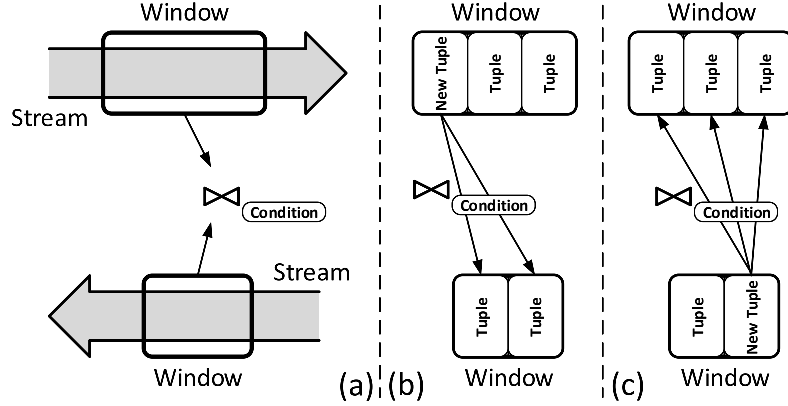


Figure 5.4.1: Sliding window concept in stream join.

they are pushed to the distribution tree one after another. They travel together (using the FIFO strategy) to reach the targeted join core. As above, here, we also assume homogeneous join cores and communication costs within the distribution tree. Then, the average visiting latency of SplitJoin is given by:

$$VL_{avg} = (T_{CL} \times log_b k) + (\frac{w}{2} \times T_P) \tag{5.3.6}$$

Thus, the average visiting latency is also logarithmic in the number of join cores, compared to the linear order in [75].

5.4 SplitJoin in Hardware

The relational join (theta join) between two non-stream relations R and S, defined as $R \bowtie_\theta S$, produces the set of all resulting pairs (r, s), which satisfy the join condition $\theta(r, s)$ and $r \in R$, $s \in S$. Extending this definition to stream join implies the same join processing semantics with the exception that streams, unlike relations, are unbounded. To mitigate the challenge of unbounded streams, with respect to both processing and storage limitations, streams are conceptually seen as bounded sliding windows of tuples, as shown in Figure 5.4.1. The size of these windows are defined as a function of time or number of tuples, referred to as time-based or count-based windows, respectively.

Figure 5.4.2 shows the traditional architecture of a join operator that receives Tuple-R and Tuple-S from streams R and S, respectively. *JC* stands for join core, which performs the join operation. To process the tuples shown in the figure, Tuple-R is inserted into *Window-R*, then it is evaluated against all existing tuples in *Window-S* and the join results are returned. Similarly, Tuple-S is inserted into *Window-S* and the same join procedure is applied.

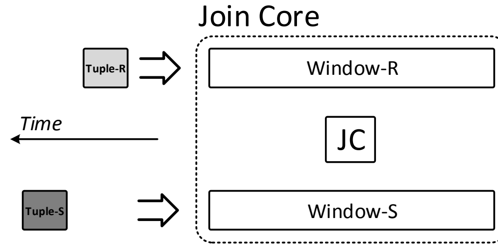


Figure 5.4.2: Traditional stream join architecture.

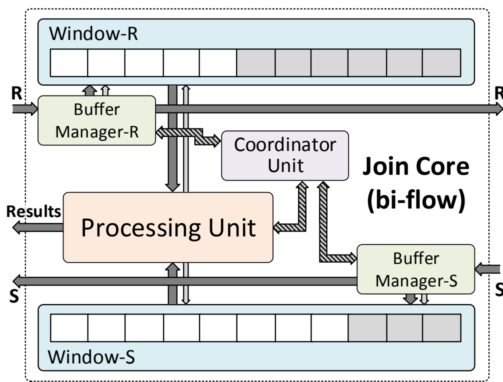


Figure 5.5.1: Bi-flow join core design.

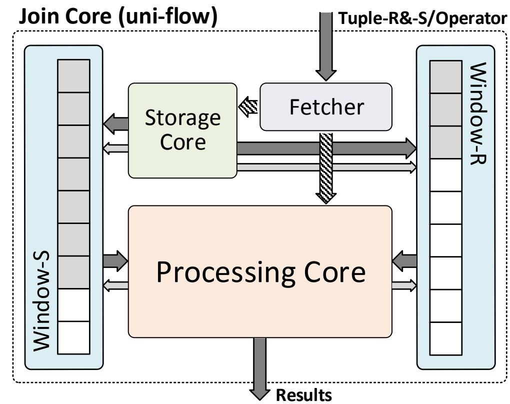


Figure 5.5.2: Uni-flow join core design.

5.5 Flow-based Stream Join in Hardware

We focus on the design and realization of a parallel and distributed stream join in hardware based on the flow-based model. Furthermore, we compare the join core internal architecture for both uni-flow and bi-flow models. Our parallel uni-flow hardware stream join architecture comprises three main parts: (1) distribution network, (2) processing (join cores), and (3) result gathering network, as shown in Figure 5.5.3.

Distribution Network —

The distribution network is responsible for transferring incoming tuples from the system input to all join cores. In this work, we present two alternatives for this network: (1) a lightweight design and (2) a scalable design.

The lightweight network distributes incoming tuples to all join cores at once without extra components, which is preferable for comparably small solutions, while the scalable variant uses a hierarchical architecture for the distribution. Here, we only present the design of the scalable network, while our experiments include evaluations for both.

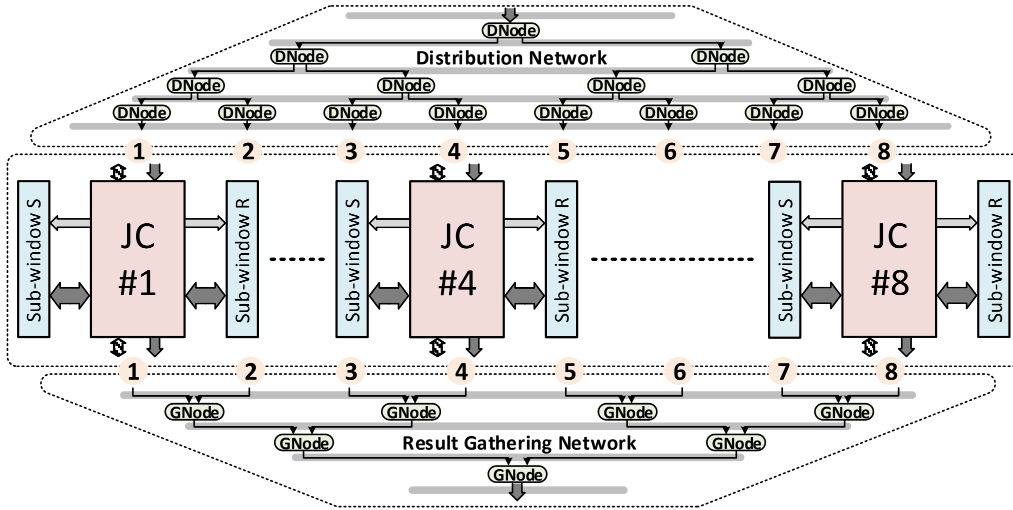


Figure 5.5.3: Uni-flow parallel stream join hardware architecture.

In the scalable distribution network, we use `DNode` to build a hierarchical network. `DNode` receives a tuple in its input port and broadcasts it to all its output ports. All `DNodes` rely on the same communication protocol, making it straightforward to scale the design by cascading them. `DNodes` store incoming tuples as long as their internal buffer is not full. As output, each `DNode` sends out the stored tuples, one tuple in each clock cycle, provided the next `DNodes` are not full. The upper part of Figure 5.5.3 demonstrates the distribution network comprised of `DNodes`. Here, we see a $1 \rightarrow 2$ fan-out size from each level to the next level from top to bottom. Other fan-out sizes (e.g., $1 \rightarrow 4$) could be interesting to explore since they reduce the height of the distribution network and lower communication latency.

The scalable distribution network consumes more resources (i.e., `DNodes`' pipeline buffers) than the lightweight variant and adds a few clock cycles, depending on its height, to the distribution latency (though it does not affect the tuple insertion throughput). On the other hand, the scalable distribution network pays off as the number of join cores increases, since it does not suffer from the clock frequency drop (degrading the performance) as observed in the lightweight design.

The `DNodes` arrangement, as shown in Figure 5.5.3, forms a pipelined distribution network. Utilizing more join cores logarithmically increases the number of pipeline stages. This means it takes more clock cycles for a tuple to reach the join cores. Nonetheless, a constant transfer rate of tuples from one pipeline stage to the next keeps the distribution throughput intact, regardless of the number of stages.

Join Cores —

In our parallel hardware design, the actual stream join processing is performed in join cores. Each

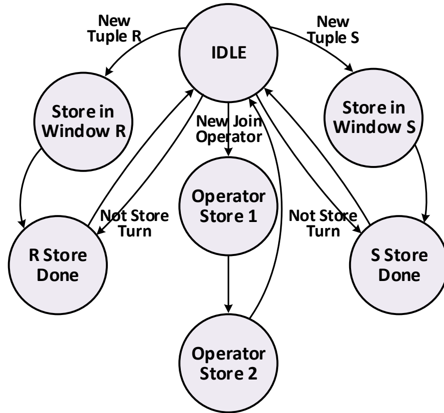


Figure 5.5.4: Storage core.

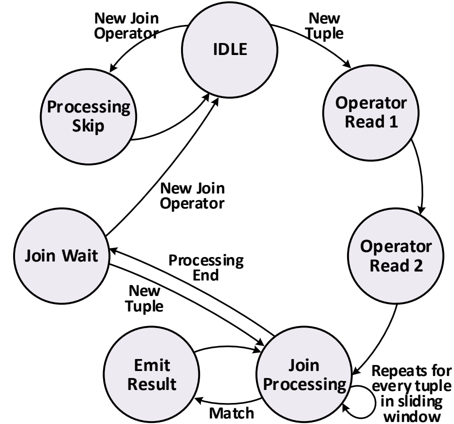


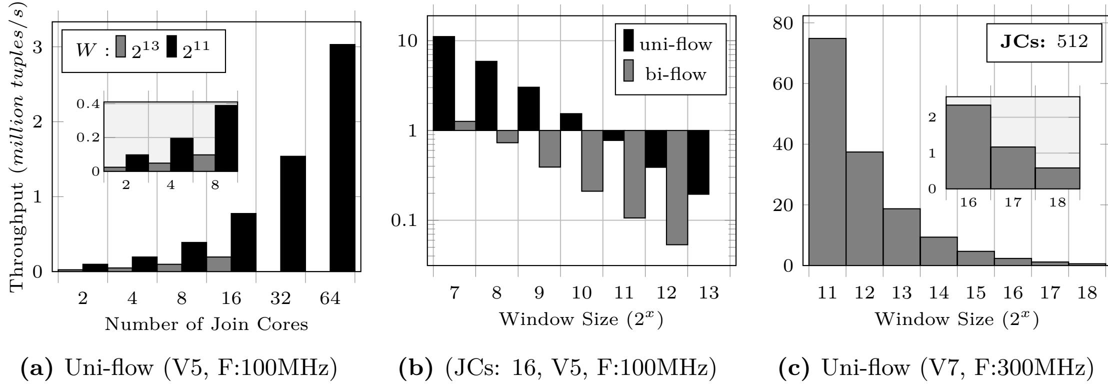
Figure 5.5.5: Processing core.

join core individually implements the original join operator (without posing any limitation on the chosen join algorithm, e.g., nested-loop join or hash join) but on a fraction of the original sliding window.

The internal architecture of our hardware join core based on the uni-flow model is shown in Figure 5.5.2, and it has `Fetcher`, `Storage Core`, and `Processing Core` as its main building blocks. `Fetcher` is an intermediate buffer that separates a join core from the distribution network. This reduces the communication path latency and improves the maximum clock frequency. The `Storage Core` is responsible for storing (and consequently expiring) tuples from R or S streams into their corresponding sub-window. The distribution task of assigning tuples to each `Storage Core` is performed in a round-robin fashion. The `Storage Core` remembers the number of tuples received from each stream and (by knowing the number of join cores and its own position in them) stores a tuple based on its turn.

The state diagram for the `Storage Core` controller is presented in Figure 5.5.4. A join operator can be dynamically programmed without the need for synthesis (individually for each join core) by an instruction which has two segments. The first segment defines join parameters such as the number of join cores and the current join core position among them, while the second segment carries the join operator conditions. The join operator programming is performed in *Operator Store 1* and *Operator Store 2*. This makes it possible to update the current join operator in real-time. After programming, `Storage Core` is ready to accept tuples from both streams. While receiving a tuple from the S stream, when it is the current join core's turn to store, the tuple is stored in its corresponding sub-window in the *Store in Window S* state; otherwise, the storage task is skipped by moving to the *S Store Done* state. The procedure for the reception of tuples from the R stream follows an identical procedure.

The `Processing Core` is responsible for the actual execution in which each new tuple (or a chunk of tuples) is compared to all tuples in the sub-window of other stream by pulling them up one at a



(a) Uni-flow (V5, F:100MHz) **(b)** (JCs: 16, V5, F:100MHz) **(c)** Uni-flow (V7, F:300MHz)

Figure 5.5.6: Throughput measurements for flow-based stream join on hardware.

time and comparing them with the newly arrived tuple.

Figure 5.5.5 presents the state diagram of the `Processing Core` controller. In the initial step it reads the join operator in *Operator Read 1* and *Operator Read 2* states. The actual comparison is performed in the *Join Processing* state, where tuples are read from their corresponding sub-window, one read per cycle. In case a match is found, it is emitted in *Emit Result*. Finally, at the end of processing, the controller waits in the *Join Wait* state for another tuple to process. After a new tuple reception, the whole execution repeats itself by means of a direct transition to the *Join Processing* state.

Result Gathering Network —

The result gathering network is responsible for collecting result tuples from join cores. Similar to the distribution network, we propose two (1) lightweight and (2) scalable alternatives for this network. We focus on the latter for descriptions while comparing both of them in the evaluation section.

The lower part of Figure 5.5.3 demonstrates the design of the result gathering network using `GNodes`. Each `GNode` collects resulting tuples from two sources connected to its two upper ports using a *Toggle Grant* mechanism that toggles the collection permission for its previous nodes in each clock cycle. As a result, each source (i.e., a join core or a previous `GNode`) pushes out a resulting tuple to the next `GNode` once every two clock cycles.

The *Toggle Grant* mechanism simplifies the design of the result collection; instead of using a two-directional handshake between two connected `GNodes` to transfer a resulting tuple, we use a single-directional signaling, in which each `GNode` looks only at permission to push out one of the results stored in its buffer. The destination (next) `GNode` simply toggles this permission each cycle without the need for any special control unit.

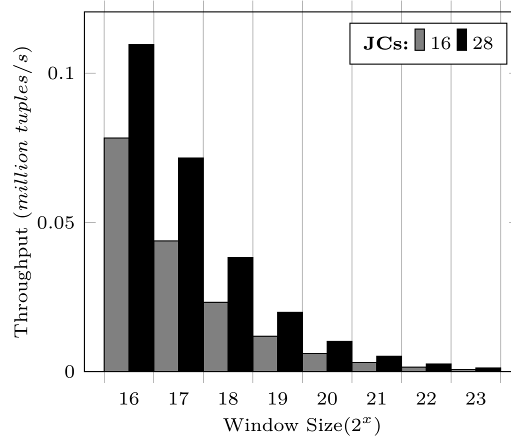


Figure 5.5.7: Throughput measurements for flow-based stream join on software.

`GNodes` configuration, as shown in Figure 5.5.3, provides a pipeline result gathering network where in each stage, tuples from two sources are merged into one and are pushed to the next pipeline stage from top to bottom (respecting our unidirectional top-down flow model). The pipelining mechanism reduces the effective fan-in size in each pipeline stage and thereby prevents clock frequency drop.

In Figure 5.5.3, arrows in the distribution and result gathering network are data buses that define the width of received tuples, including their 2-bit headers. The header defines whether we are dealing with a new join operator or a tuple belonging to either the R or S stream. The width of the data bus for result tuples is twice (not counting the header) the size of the input data bus since a result is comprised of two input tuples that have met the join condition(s).

Flow-based Hardware Design Comparison —

In the uni-flow model, data passes in a single top-down flow, in which each join core receives tuples directly from the distributor and operates independently from other join cores (Figure 5.1.1b). The uni-flow design offers full utilization of the communication channel's bandwidth, specifically from the input to each join core, since all tuples travel over the same path to each join core. Therefore, regardless of the incoming tuple rate for each stream, every tuple has access to the full bandwidth. To clarify this issue for the bi-flow model, assume we are receiving tuples only from stream R; then, all communication channels for stream S are left unutilized. Even with an equal tuple rate for both streams, it is impossible to achieve simultaneous transmission of both T_R and T_S between two neighboring join cores due to the locks needed to avoid race conditions. Furthermore, comparing the internal design of a join core based on the bi-flow model, Figure 5.5.1, and one based on uni-flow Figure 5.5.2, we see a significant reduction in the number of internal components that correspondingly reduces the design complexity. Neighbor-to-neighbor tuple traveling circuitries for two streams are eliminated from `Buffer Manager-R & -S` and `Coordination Unit`, as they are reduced and merged to form `Fetcher` and `Storage Core` in the join core based on the uni-flow

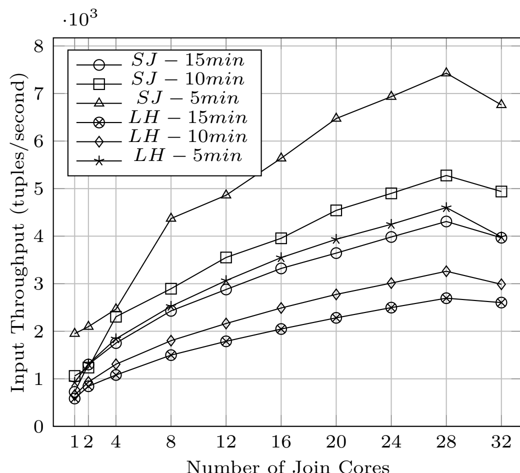


Figure 5.6.1: Throughput comparison.

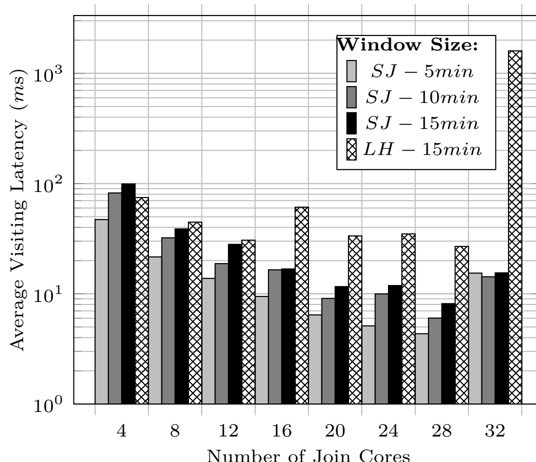


Figure 5.6.2: Visiting latency comparison.

model. This improvement also reduces the number of I/Os from five to two, which significantly reduces the hardware complexity, as the number of I/Os is often an important indication of complexity and final cost of a hardware design.

5.6 Experimental Results

In this section, we experimentally evaluate our SplitJoin implementation. All experiments are performed on a 32-core system. Our system is a Dell PowerEdge R820 featuring 4 × Intel E5-4650 processors and 32 × 16GB DDR3 memory (RDIMM, 1600 MHz, Low Volt, Dual Rank, x4). We ran our benchmarks on Ubuntu 14.04.2 LTS (GNU/Linux 3.13.0-57-generic x86_64) installed on a Docker container [74] running on the same host OS.

5.6.1 Experimental Setup

We adopted the benchmark used in recent stream join approaches [39, 85, 75]. In this benchmark two streams $R =$ `(x:int, y:float, z:char[20])` and $S =$ `(a:int, b:float, c:double, d:bool)` are joined via the two-dimensional band join, as follows:

```
WHERE r.x BETWEEN s.a-10 AND s.a+10
  AND r.y BETWEEN s.b-10 AND s.b+10
```

In our evaluations, we used the low-latency (referred to as LH vs. SplitJoin (SJ)) and the original

handshake join libraries that were kindly provided by the authors of [85, 75]. Also in line with related approaches, integers and floats were generated following a uniform distribution in the range of $1 - 10^4$, unless otherwise stated.

The results cover the end-to-end evaluation, including data distribution network, SplitJoin storage and processing cores, the result gathering network, and also the proposed punctuated ordering mechanism. The punctuation precision is based on the outer tuple ordering as shown in Figure 5.2.1, unless otherwise stated.

In our time-based window realization, we generated timestamps on-the-fly using the system call `clock_gettime()`. Using a synthetic timing mechanism, as we experimented, further improves the overall performance by about 15% by relieving the overhead incurred by system calls.

5.6.2 Performance and Scalability

We evaluate SplitJoin performance by measuring latency and throughput metrics as we scale the level of parallelism. In general, key factors that influence the stream join performance are how the input streams are flowing through the join cores and how the joined results are collected and flow to the output.

In Figure 5.6.1, we demonstrate throughput results of SplitJoin in comparison with [75]. As we scale the number of join cores, we observe that both solutions scale gracefully; however, SplitJoin outperforms the low-latency handshake join by up to 60% (comparison between 15-min sliding windows). Theoretically, the performance of both approaches should be similar, as both utilize all join cores in parallel to process incoming tuples. However, the core-to-core communication and mandatory expiry messages in low-latency handshake join (necessary for both time-based and count-based join versions) impose a noticeable penalty.

In Figure 5.6.1, we also observe how the two approaches perform for different time-based window sizes. When the join core count is 32, we observe a drop in performance in both of the approaches. This is due to the existence of extra threads to perform other (non-processing) tasks such as stream distribution and result gathering in case of SplitJoin, and tuple assignment, expiry message generation, and result gathering in case of the low-latency handshake join. Since our system has only 32 processing cores, by instantiating 32 join cores, the operating system is forced to perform context switches, resulting in system saturation and performance drop.

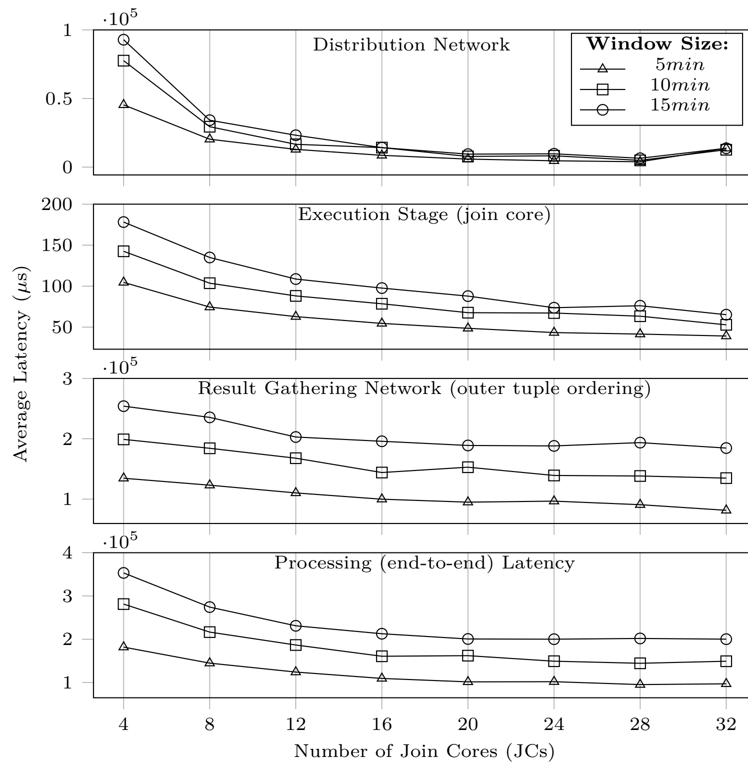


Figure 5.6.3: SplitJoin latency measurements.

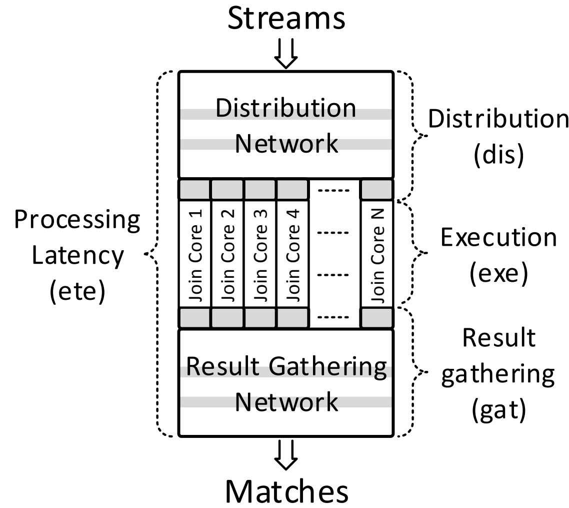


Figure 5.6.4: SplitJoin processing pipeline.

5.6.3 Latency Evaluations

In Figure 5.6.4, we present an abstract model of our end-to-end processing pipeline stages. The grayed parts show intermediate and pipeline buffers. In the measurements, we are reporting the latency of the distribution stage (*dis*), which also includes the time that tuples are waiting in the pipeline stage between the distribution and execution stages. The latency of the execution stage (*exe*) is the latency attributed to the time that it takes a tuple to pass through the processing and storage steps in the join core, which also includes the tuple expiration process for the time-based sliding window. The latency for the last stage includes the time that resulting tuples are waiting in the pipeline stage between the execution and result gathering stages and also the time for the *Merger* and the *Collector* units to bring them to the output of SplitJoin. Latency reports for these measurements plus the processing, end-to-end (*ete*), and latency of SplitJoin, for the time-based sliding window, are presented in Figure 5.6.3.

Processing Pipeline Stage Latency —

In the distribution network, as we increase the number of join cores, incoming tuples are distributed between larger number of join cores instead of having to pile up in the pipeline buffer for fewer join cores. Therefore, increasing the number of join cores, inherently reduces the waiting time in the distribution stage as shown in Figure 5.6.3.

Since our evaluation system has only four processor sockets, the increase in the size of the distribution network has no significant effect on the performance except when the size of the sliding window is small. In the execution stage, the increase in the number of join cores for a given window size

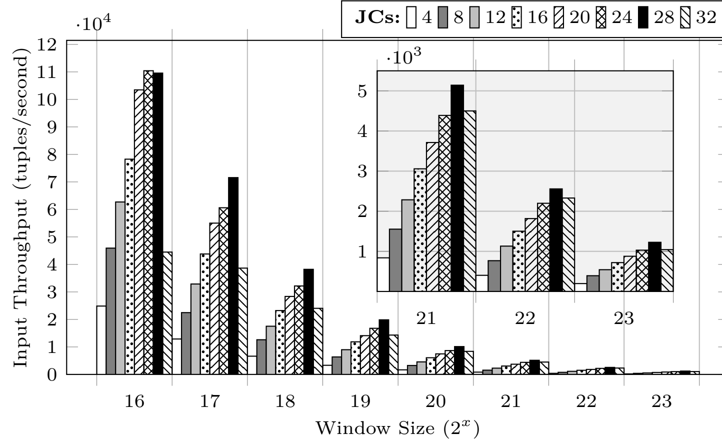


Figure 5.6.5: Count-based SplitJoin throughput.

translates into smaller sub-windows for each join core and the latency also proportionally decreases.

Among our three pipeline stages, the result gathering network with punctuation ordering had the highest latency impact. This latency was mainly due to the waiting times of the *Mergers* on one of their input ports to receive a punctuation mark (*star*) before starting to read from their next port.

Visiting Latency: In Figure 5.6.2, we observe the average visiting latency ($T_{match} - max(t_r, t_s)$) for SplitJoin with 5, 10, and 15 minutes sliding windows, and low-latency handshake join with a 15 minutes sliding window for varying number of join cores. The t_r and t_s stand for initial timestamp of r and s tuples, respectively.

As we evaluated the average visiting latency (cf. Section 5.3), the latency increases logarithmically, $O(\log_b k)$, for SplitJoin as opposed to linearly, $O(k)$, for the low-latency handshake join [75]. By comparing the average visiting latency for the 15-min version of SplitJoin and low-latency handshake join, when we use four join cores, the latency is quite similar; however, once the number of join cores increases, the gap between SplitJoin and low-latency handshake join widens drastically by a factor of up to 3.3X (8.1ms vs. 26.8ms for 28 join cores).

We observe an increase in latency while reaching 32 join cores which is again due to the lack of enough resources for the other (non-processing) tasks. Since low-latency handshake join requires to perform additional costly tasks, such as emitting individual expiry message for each tuple, the resource contention shows a more significant impact on latency as seen when instantiating 32 join cores.

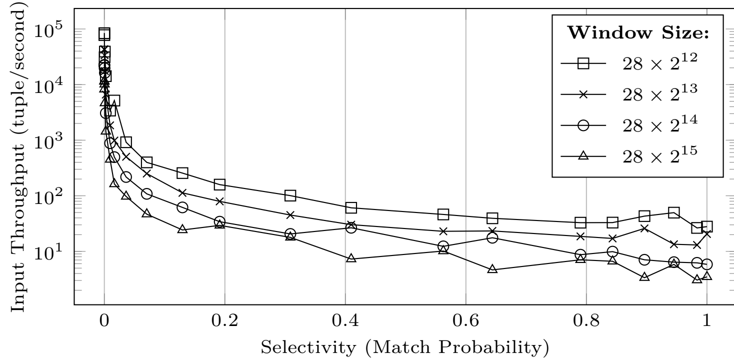


Figure 5.6.6: Selectivity effect on SplitJoin throughput (28 JCs).

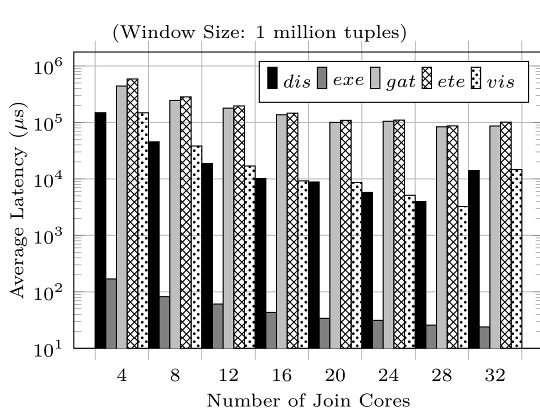


Figure 5.6.7: Count-based SplitJoin latency reports (uniform distribution:$1 - 10^5$).

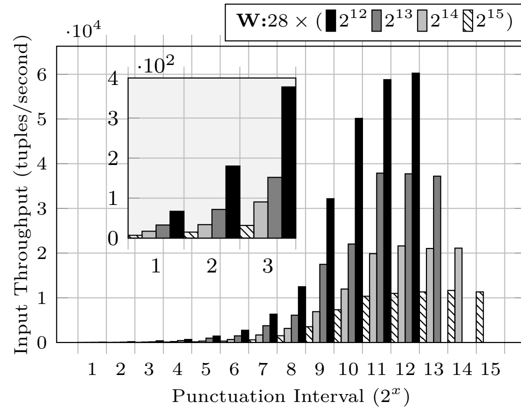


Figure 5.6.8: Ordering precision effect (28 JCs, uniform distribution:$1 - 10^4$).

5.6.4 Count-based Sliding Window

Although the count-based and time-based versions of SplitJoin behave similarly, there are two key differences: (1) having no space allocated for timestamp values and no on-the-fly generation of timestamps through a costly system call and (2) having a fixed window size for count-based semantics as opposed to the time-based semantics where the window size varies depending on the incoming tuple rate. These differences result in roughly 20% improvement in performance for SplitJoin using a count-based instead of time-based sliding window. For example, SplitJoin instantiated with 28 join cores over a 15-min sliding window (shown in Figure 5.6.1) sustains an input rate of 4400 tuples/second, which roughly translates to window sizes of 2^{21} for each stream. But for the count-based window, if we set the window size to (2^{21}), SplitJoin can process up to 5200 tuples/second, as shown in Figure 5.6.5.

In the count-based results shown in Figure 5.6.5, we observe two effects: (1) larger window sizes result in fewer punctuation marks, assuming the same input throughput, since in the outer tuple

ordering each join core produces one punctuation mark at the end of each tuple processing and (2) a larger sub-window per join core additionally increases the processing efficiency by reducing the impact of other (non-processing) pipeline stages. Based on these observations, doubling the window size while fixing the number of join cores reduces the processing throughput.

For each window size, by increasing the number of join cores (**JCs**), we observe a relative improvement in the throughput except when using 32 join cores. Over-utilizing system resources (i.e., using 32 join cores) has more impact on the throughput for smaller window sizes. Larger windows keep join cores busier, thus, new tuples are processed after longer waits. This relieves other tasks (i.e., distribution), reducing the effect of resource contention.

In Figure 5.6.7, we present the latency of the processing pipeline stages, the average processing latency (*ete*), and visiting (*vis*) latency for SplitJoin for the count-based sliding window. SplitJoin scales gracefully as we increase the number of join cores; in particular, using 28 join cores, the visiting latency is improved by more than 2.5X and 8.3X as compared to the time-based version of SplitJoin and the low-latency handshake join, respectively.

5.6.5 Effect of Selectivity

The selectivity (also called match probability) is one of the major factors affecting join performance. Often a low selectivity is assumed in most related work [39, 85, 75]. However, it is important to analyze the sensitivity of a join algorithm with respect to the selectivity in order to assess the generality of the approach.

The effect of varying the selectivity on the input throughput is illustrated in Figure 5.6.6. The key observation is that SplitJoin's latency scales reasonably, and it is robust to changes of selectivity, even for sliding windows as large as 28×2^{15} tuples.

5.6.6 Effect of Punctuation Precision

Figure 5.6.8 demonstrates the effect of the ordering precision on the processing performance. In this diagram, we utilize 28 join cores with varying sub-window sizes ($2^{12} - 2^{15}$) per join core. The ordering precision starts from one punctuation per sub-window processing, referred to as *relaxed inner ordering*, and progressively increases the precision until one punctuation mark (*star*) is produced after each comparison (represented as 2^1 on the x-axis) within each sub-window, referred to as *strict inner ordering*.

The *relaxed inner ordering* is the same as the *strict outer ordering*. Therefore, the highest

punctuation interval for each window size in Figure 5.6.8 represents the effect of *strict outer ordering* on the throughput for that window size.

As we increase the precision (e.g., focusing on a sub-window size of 2^{15}), from $2^{11} - 2^{15}$, its effect on the overall performance is negligible, since the number of punctuations produced per each incoming tuple in each join core is relatively low (i.e., 1, 2, 4, 8, and 16 punctuations, respectively) compared to the sub-window size which is 2^{15}. However, as we continue to increase the precision from the interval 2^{10} down to 2^1, the number of punctuations becomes comparable to the sub-window size for each join core, and as expected, negatively affects the performance of SplitJoin. This highlights the importance of balancing ordering precision versus overall performance. In fact, since the precision is adjustable, to achieve a desired throughput, SplitJoin could adaptively adjust the precision interval to achieve a sweet spot between the ordering precision and the sustainable input throughput.

5.6.7 Hardware Splitjoin Evaluations

For hardware experiments, we synthesized and programmed our solution on a ML505 evaluation platform featuring a Virtex-5 XC5VLX50T FPGA. Additionally, we synthesized our solution on a more recent VC707 Evaluation board featuring a Virtex-7 XC7VX485T FPGA. For software experiments, we used a 32-core Dell PowerEdge R820 featuring 4 × Intel E5-4650 (TDP: 130 Watt) processors and 32 × 16GB (PC3-12800) memory, running Ubuntu 14.04.2 LTS.

We realized parallel stream join based on bi-flow model using a simplified OP-Chain topology from FQP[2], proposed in [67]. We used the `Xilinx` synthesis tool chain to synthesize, map, place, and route both of the bi-flow and uni-flow parallel hardware realizations and loaded the resulting bit file onto our FPGA using a JTAG interface. This bit file contains all required information to configure the FPGA.

The input streams consist of 64-bit tuples that are joined against each other using an equi-join, though there is no limitation on the condition(s) used. Both of the realizations have the ability to adopt larger tuples that are defined by pre-synthesis parameters.

Throughput Evaluation —

Throughput measurements for bi-flow and uni-flow parallel stream join realizations are presented in Figure 5.5.6. For the uni-flow version, we were able to instantiate 16 join cores on our platform with up to $W : 2^{13}$ window size (per stream), as we see in Figure 5.5.6a. We observe a linear speedup with respects to the number of join cores as expected. We were not able to realize window sizes larger than 2^{11} when instantiating 32 and 64 join cores due to the extra consumption of memory resources in the distribution and result gathering networks and auxiliary components.

[2] FQP is available in VHSIC Hardware Description Language (VHDL).

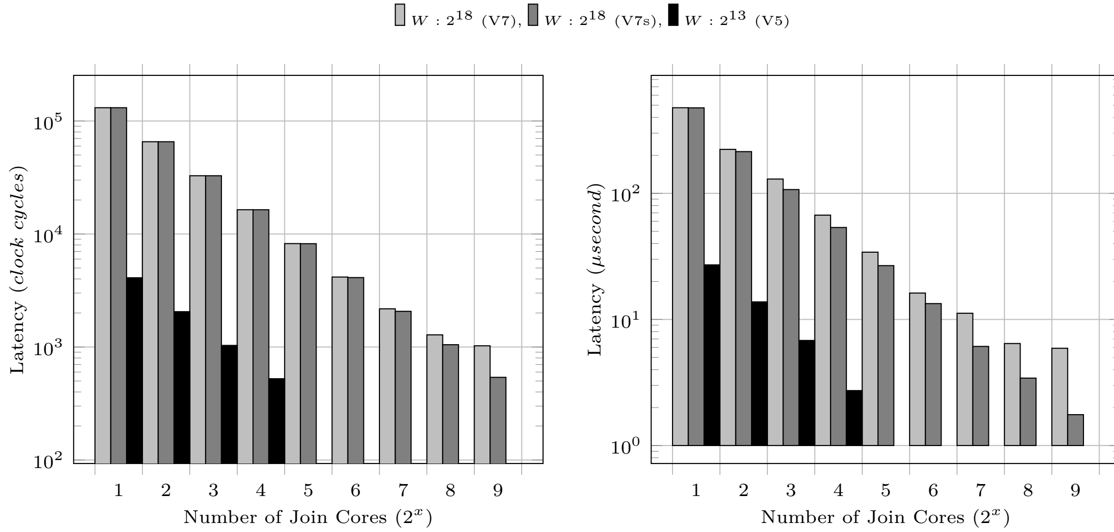


Figure 5.6.9: Latency reports for uni-flow parallel stream join on hardware.

Figure 5.5.6b presents the comparison between the input throughput in a parallel stream join based on uni-flow and bi-flow models as we change the window size. We observe nearly an order of magnitude speedup when using a uni-flow compared to a bi-flow model. Although in theory, both models are similar in their parallelization concept, the simpler architecture in uni-flow brings superior performance. We were not able to instantiate 16 join cores with 2^{13} in bi-flow hardware, unlike the uni-flow one, because each join core is more complex and requires a greater amount of resources to realize it.

Figure 5.5.6c presents extracted (from a synthesis report) throughput on a mid-size, but more recent, Virtex-7 (XC7VX485T) FPGA. We were able to realize a uni-flow parallel stream join with as many as 512 join cores and window sizes as large as 2^{18}. We used a 300MHz clock frequency for this evaluation as provided by the synthesis report. As a result of having more join cores and a higher clock frequency, we see acceleration of around two orders of magnitude when we utilize a window size of 2^{13} compared to the realization on Virtex-5 (Figure 5.5.6a).

We ran our experiments on the software realization of a uni-flow parallel stream join (available from [68]) and the throughput results are presented in Figure 5.5.7 for 16 and 28 join cores. Similar to the experimental setup in [68], the maximum input throughputs were achieved while using 28 cores out of 32 on our platform since some internal components in SplitJoin, i.e., the distribution and result gathering network, also consume a portion of the processors' capacity.

Although the operating frequency of the Virtex-7 FPGA is significantly lower than that of the processors used in our system, 300MHz compared to the processor base frequency of 2.7GHz and max turbo of 3.3GHz, still we observed around 15× acceleration compared to the software realization (28 join cores) while using the same window size (2^{18}) on both platforms (Figures 5.5.6c

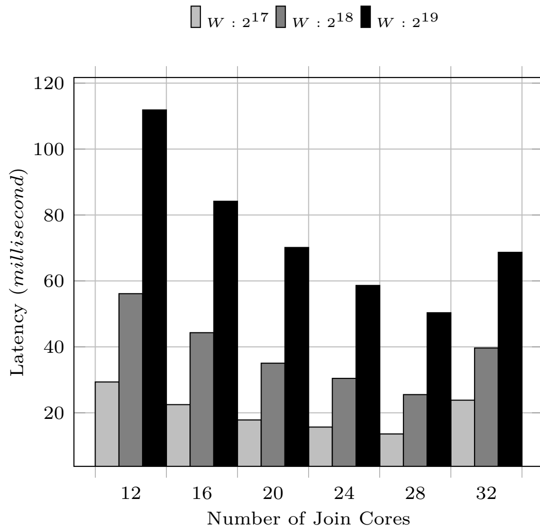


Figure 5.6.10: Latency reports for uni-flow design on software.

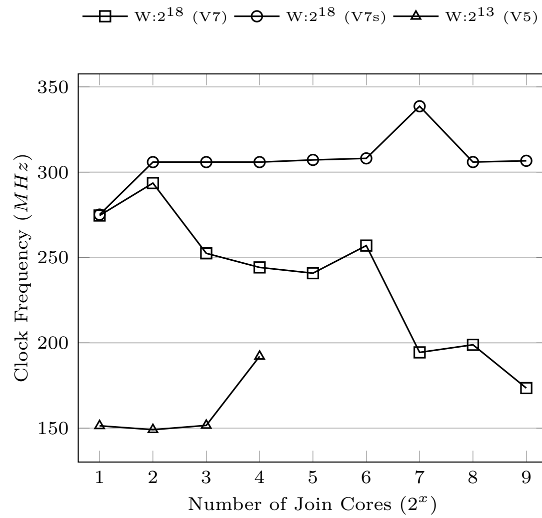


Figure 5.6.11: Uni-flow design clock frequencies on Virtex-5 and Virtex-7.

and 5.5.7). Two factors are main contributors to this throughput gain: 1) ability to instantiate more join cores compared to the software version, since they operate in parallel and linearly increase the processing throughput as a function of their count. 2) utilizing internal BRAMs in FPGA by essentially coupling data and processing in each join core, while in the software variant, the sliding window data resides in the main memory. This data has to move back-and-forth through the memory hierarchy for each incoming tuple.

Latency Evaluation —

We refer to latency as the time it takes to process and emit all results for a newly inserted tuple. We mainly focus on latency comparisons between the hardware and software realizations of the uni-flow model for a parallel stream join. The measurements for these comparisons are shown in Figures 5.6.9 and 5.6.10.

Figure 5.6.9 captures the latency observed with respects to the number of clock cycles and the execution time (μsecond). For realization on Virtex-5 ($V5$), we used the lightweight distribution and result gathering networks since the system is relatively small; however, for the synthesized design on Virtex-7 ($V7$), we have reports for both lightweight and scalable (specified by s in figures) variants of distribution and result gathering networks.

As we increase the number of join cores, we do not observe a significant difference in the number of cycles required to process a tuple in either realization. The distribution network in the lightweight design requires fewer cycles to transfer incoming tuples to all join cores while on the scalable version, a tuple has to travel through multiple distribution levels ($\log_2 N$, where N is the number of join

cores) to reach join cores; however, this advantage is neutralized by the greater latency in the lightweight result gathering network. The cost of round-robin collection from join cores, one after another, quickly becomes dominant as we approach larger numbers of join cores. However, by taking into account the clock frequency drop in the lightweight solution as we increase the number of join cores, the actual difference in latency becomes significant, as shown in Figure 5.6.9.

Utilizing a window size of 2^{18} for each stream, the hardware version (Figure 5.6.9) shows around two orders of magnitude improvement in latency compared to the software variant (Figure 5.6.10), mainly due to massive parallelism and memory and processing coupling.

Scalability Evaluation —

The scalability of a hardware design is determined by how the maximum operating clock frequency is affected as we scale up the system. Here we scale up our solution by increasing the number of join cores that translates to a linearly increase in the processing throughput.

Figure 5.6.11 shows how clock frequency changes as we increase the number of join cores for lightweight versions on Virtex-5 (V5) and Virtex-7 (V7) and scalable version on Virtex-7 (V7s). For the realization on our Virtex-5 FPGA, we do not see any significant drop as we increase the number of join cores. Although we are using the lightweight version, the system size (number of join cores) is rather small to show its effect; actually, we even see an increase in the clock frequency when utilizing 16 join cores that is due to heuristic mapping algorithms adopted by the synthesis tool[3].

For larger uni-flow based realization with more join cores, we see how the clock frequency of the lightweight version drops as we increase the number of join cores. Since the Virtex-7 FPGA supports higher clock frequencies, compared to the Virtex-5, it is more sensitive to large fanout sizes and longer signal paths; therefore, we see this effect even when using 8 and 16 join cores. For the hardware realization based on uni-flow with scalable distribution and result gathering networks, we observe no significant variations in the clock frequency as we scale up the system.

Power Consumption Evaluation —

The extracted power consumption reports when using 16 join cores with a total window size of 2^{13} (for each stream) consumed $1647.53mW$ and $800.35mW$ power for parallel stream join based on bi-flow and uni-flow, respectively. As expected simpler design and correspondingly smaller circuit size resulted in more than 50% power saving in utilizing uni-flow compared to bi-flow.

[3]Using more restrictions (such as using a higher clock constraint, *e.g.*, 190MHz) it is possible to achieve higher clock frequencies when utilizing fewer number of join cores.

Chapter 6

Scalable Multiway Stream Joins in Hardware

Efficient real-time analytics are an integral part of an increasing number of data management applications, such as computational targeted advertising, algorithmic trading, and Internet of Things. In this section, we primarily focus on accelerating stream joins, which are arguably one of the most commonly used and resource-intensive operators in stream processing. We propose a scalable circular pipeline design (Circular-MJ) in hardware to orchestrate a multiway join while minimizing data flow disruption. In this circular design, each new tuple (given its origin stream) starts its processing from a specific join core and passes through all respective join cores in a pipeline sequence to produce the final results. We also present a novel two-stage pipeline stream join (Stashed-MJ) that uses a best-effort buffering technique (referred to as stash) to maintain intermediate results. If an overwrite is detected in the stash, our design automatically resorts to recomputing intermediate results. Finally, we present a parallelized version of our multiway stream join by integrating our proposed pipelines into a parallel unidirectional flow-based architecture (Parallel-MJ). Our experimental results demonstrate a linear throughput scaling with respect to the numbers of streams and processing cores.

This section presents the following contributions:

1. We propose a scalable multiway stream join (Circular-MJ) on hardware that is built on a circular chain of dedicated stages (one per stream) and that benefits from pipeline parallelism.

2. We present a novel two-stage pipeline (Stashed-MJ) that benefits from a stash (intermediate results buffer) to accelerate processing throughput.

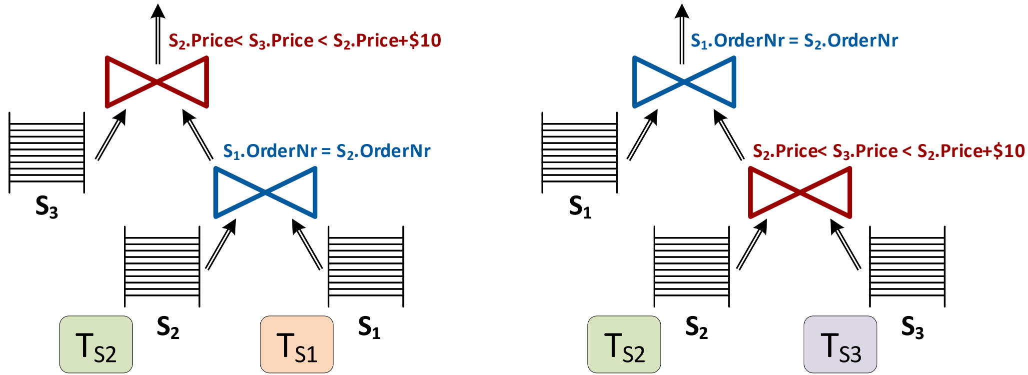


Figure 6.1.1: Reordering operators in multiway stream joins.

3. We enhance our multiway stream join pipeline to integrate it into the parallel architecture to linearly scale the processing capabilities of our solution (Parallel-MJ).

6.1 Multiway Stream Join

A conventional join operates on tuples originating from two sources. Naturally, we should be able to cascade the join operators to support more than two sources (streams). However, avoiding arbitrary (unstructured) communications between processing components, which is a crucial property for hardware design, introduces the challenge of real-time join operator reordering, as demonstrated in Figure 6.1.1. Here, tuples from streams S_1 and S_2 keep the order of join operators intact (left figure), while a new tuple from stream S_3 requires operator reordering (right figure). Without the reordering, we have to recompute all intermediate join results between all existing tuples in the sliding windows of S_1 and S_2, which is not feasible due to the size and complexity of this processing. The reordering challenge is exacerbated when working with a hardware system, where changes in the data path and control circuitry of a design, particularly as it is scaled up (i.e., in the number of streams), have severe effects on the design complexity, performance, and cost of the system.

Each new tuple insertion into the multiway stream join updates its sliding window and subsequently may produce new intermediate results. Without materializing the intermediate results, we need to sequentially cascade the join operators and feed each new tuple always from the bottom (input port) of the cascaded architecture. Each new tuple and subsequently its intermediate results pass through all join operators for processing, which leads to a right-deep join tree architecture.

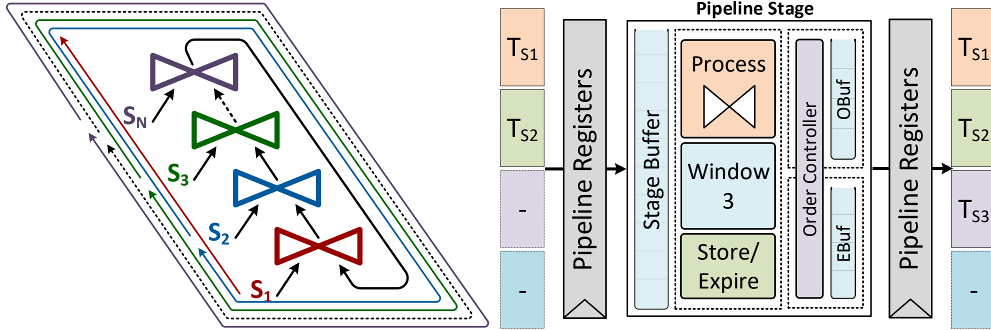


Figure 6.1.2: Multiway stream joins with circular design.

Figure 6.1.3: Intermediate result generation in a pipeline stage.

6.1.1 Multiway Join with a Circular Pipeline

Designing hardware based on join reordering poses a scalability issue due to the required crossbar connection[1] between processing units (join operators) and sliding windows.

To address this issue, we need to fix the order of join operators, which hardwires each sliding window to only one processing unit. This approach eliminates the need for a crossbar. To fix each join operator's location on the right-deep join tree, we propose a circular data path design that connects all operators together, as shown in Figure 6.1.2. In this design, each join operator is connected to only one sliding window with two entries. Each operator receives its new tuples, determined based on their origin, from its right entry. The left entries are placed in the closed circular path, which carries the intermediate results from one join operator to the next join operator. The resulting tuples are emitted after processing a new tuple in exactly $N - 1$ operators, where we have N streams. The remaining operator is responsible for storing the new tuple in its sliding window.

Using this design (Figure 6.1.2), we propose a scalable circular pipeline for multiway stream joins in hardware (referred to as Circular-MJ), as shown in Figure 6.1.4. This pipeline has the same number of stages as input streams. Each stage is placed between two isolating sets of registers and is responsible for processing a new tuple against a specific sliding window. If the window in a stage belongs to the current tuple's origin, store and expiration tasks are performed rather than the processing. To handle data transfer between stages in a scalable manner, we arrange the stages in a circular architecture such that the intermediate results of each stage are fed to another stage as input.

[1] A type of connection that provides the possibility for every input to access all output ports.

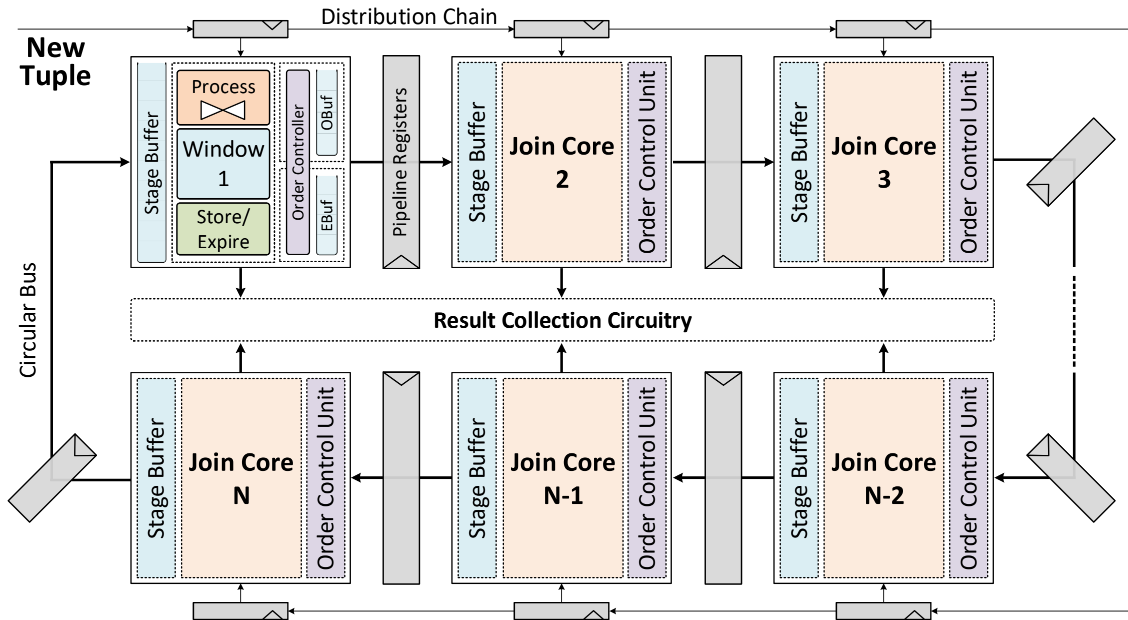


Figure 6.1.4: Circular-MJ architecture.

6.1.2 Circular Pipeline Design Rationale

The decision that has heavily influenced our design is to have an independent operation on each window, assuming that we already have the intermediate results from another join operator(s). Therefore, we design N stages, where each stage is responsible for processing, storage, and expiration operations on a single sliding window. To handle data transfer between stages in a scalable manner, we arrange the stages in a circular architecture such that the intermediate results of each stage are fed to another stage as input.

The key intuition of our circular design is that rather than adapting the order of join operations to incoming tuples, we adapt the insertion location for incoming tuples using a pipelined distribution chain. In other words, rather than having single entry and reordering join operators, we change the architecture to have an adaptive entry to fix the order of join operators and to respectively hardwire each sliding window to a separate processing unit (see Figure 6.1.4).

Our circular design differs from a conventional pipeline in three aspects: (1) a circular bus that passes through all stages within the pipeline that relies exclusively on direct neighbor-to-neighbor communication; (2) arbitrary access to any stage in the pipeline rather than physically fixing each stream to one join operator in the pipeline; (3) arbitrary output collection from all stages, which is necessary to offload the final results while sustaining high throughput.

6.1.3 Circular-MJ Architecture

In our circular pipeline, each stage is connected to its two neighboring stages using the *circular bus*, as shown in Figure 6.1.4. This bus provides a processing path that encompasses all pipeline stages. Each stage contains three main components: (1) a stage buffer, (2) a join core, and (3) an order control unit. To feed the incoming tuples to the stages, we use a pipelined chain of registers (referred to as a *distribution chain*) that carries new tuples to their corresponding stage. The purpose of the distribution chain is to keep the circular pipeline at full utilization and maximize processing throughput.

(1) The **stage buffer** collects intermediate results from a previous stage and feeds these results one-by-one to its stage's join core. The existence of this buffer is necessary to prevent stalls caused by the joint burst of intermediate results and new tuples. A stall may occur due to low selectivity (high match probabilities[2]) in multiple consecutive join operators in the deep join tree, which leads to the generation of many intermediate results.

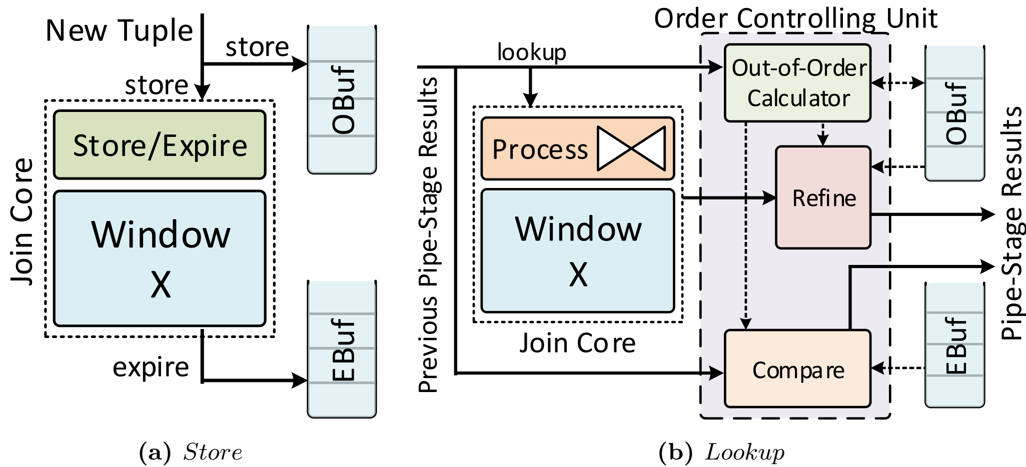
In the stall scenario, all stage buffers are full, and each stage is waiting for the next stage to consume some of the intermediate results in its buffer for further processing. Consider this scenario from the perspective of the ith stage, which is waiting for the $(i+1)^{th}$ stage to consume some of the intermediate results in its stage buffer. Then, given N streams, the nth stage waits for the 1st stage, and this dependency reaches the $(i-1)^{th}$ stage, which is waiting for the ith stage since the pipeline stages are arranged in a circular architecture. Therefore, the ith stage is waiting for itself to continue the processing, which translates to a stall. Using a stage buffer at the entry of each stage prevents stalls from occurring by providing extra space to store intermediate results. In this way, the ith stage can push its produced intermediate results without waiting for the next stage to consume the results.

Furthermore, we give priority to intermediate results over new tuples, which further reduces the probability of a stall occurring. This means that when there is a choice between processing an intermediate result or a new tuple, the stage controller selects the former to reduce the stage buffer's fill ratio. Additionally, to prevent stalls, we drop an intermediate result when the stage buffer fill ratio exceeds a specified threshold. However, this can lead to producing only a subset of the join results.

(2) The **order control unit** is responsible for governing the correct execution order, considering the out-of-order tuple insertion into the processing pipeline by the distribution chain.

Each new tuple is stored in its corresponding sliding window while other join cores are still (concurrently) processing (computing the join) previously received tuples and their intermediate

[2]The probability that any two consecutive tuples (each from one of the streams) satisfy a join condition. Thus, as *selectivity* increases, the *match probability (mp)* decreases.



(a) *Store* **(b)** *Lookup*

Figure 6.1.5: Order control unit architecture.

results. This is referred to as the out-of-order tuple insertion in our solution. Without handling this case, the intermediate results from the previously inserted tuples are considered against an incorrect set of tuples that can lead to missing or extraneous results.

This unit becomes involved in two tasks of a new tuple store and lookup, as shown in Figure 6.1.5. This unit has an order buffer (OBuf) that stores newly received tuples from the distribution chain and an expiry buffer (EBuf) that stores expired tuples from the sliding window. Having the uniform distribution for the tuples' origin (stream number), we suggest that the size of OBuf and EBuf be the same as the number of streams. This results in better utilization of the join cores.

The order control unit has an *out-of-order calculator*, which counts the number of tuples that belong to its stage's sliding window that are stored in this window prematurely[3]. To avoid incorrect matches with prematurely stored tuples, the order control unit uses a *refine* component to drop these tuples. To avoid missing matches with prematurely expired tuples, a *compare* component performs an additional comparison of the same number of prematurely stored tuples with the last expired tuples in the EBuf.

(3) The **join core** contains its dedicated stream's sliding window in addition to the processing, storage, and expiration components that operate on this window. The sliding window can be count- or time-based, and the joining algorithm can encompass different approaches ranging from the nested loop approach used for general joins to the hash-based approach used for equi-joins. In this work, we mainly focus on the count-based sliding window, which requires more complex control logic to guarantee the correct execution order. In the case of having time-based sliding windows, the timestamps inherently filter out extra results, while the expiration from each window must be delayed to ensure that there are no missing results.

[3]Earlier than their order corresponding to the current tuple under processing.

6.1.4 Circular-MJ Operation

According to the join operator placement (assuming a linear mapping of $S_{1..n}$ to join cores $1..N$) in Circular-MJ a new tuple from S_i starts processing from the pipeline stage $i+1$ to face its corresponding join condition. With a circular bus, the processing of a new tuple from stream S_n (T_{S_n}) starts from the 1st stage. Starting from an earlier stage, we need to have the *union* of the new tuple with all tuples in that stage's sliding window since there is no condition between them. This is impractical due to the potentially large number of intermediate results from the union operator.

Each new tuple is transported to each pipeline stage by the distribution chain. As demonstrated in Figure 6.1.4, each distribution chain pipeline buffer has one input and two output ports (except for the last one). When this buffer is empty, it accepts a new tuple from its previous buffer and holds this new tuple in both of its output ports. Once the new tuple is consumed by the join core and the next buffer is also empty (ready to receive a new tuple), the current buffer pushes this new tuple to the next buffer and waits to receive another new tuple. The last buffer in the chain drops its tuple as soon as it is consumed by the last join core.

Join Core Lookup and Storage —

After processing in one stage, the intermediate results are pushed to the next pipeline stage for the next processing step. All join cores temporarily receive a copy of each new tuple, but only the join core that is home to the new tuple's sliding window stores this tuple.

There are three possibilities when a new tuple enters a pipeline stage. In the first case, the new tuple belongs to neither the current stage nor the previous stage's sliding windows. Here, the new tuple is temporarily stored in the OBuf without triggering any further operations. In the second case, the new tuple belongs to this stage's sliding window, and it is stored there. The new tuple is also stored in the OBuf, while the expired tuple from the sliding window is stored in the EBuf, as shown in Figure 6.1.5a. In the third case, the new tuple belongs to the sliding window of the previous stage. Here, the new tuple is stored in the OBuf and triggers the processing. The generated intermediate results are pushed (one-by-one) to the next stage for further processing.

The circular bus that carries the intermediate results between the pipeline stages has a reserved field for each stream, and each stage fills its corresponding field with the tuple that has met its join condition. When a stage fills the last reserved field, the final result is pushed out from the join core to the *result collection circuitry*, as shown in Figure 6.1.4, to transfer to the system's output. The reserved fields and an example of filling one of them are shown in Figure 6.1.3. In this example, the join condition could be between streams S_2 (or S_1) and S_3.

Join Core Lookup and Store/Expire Controller —

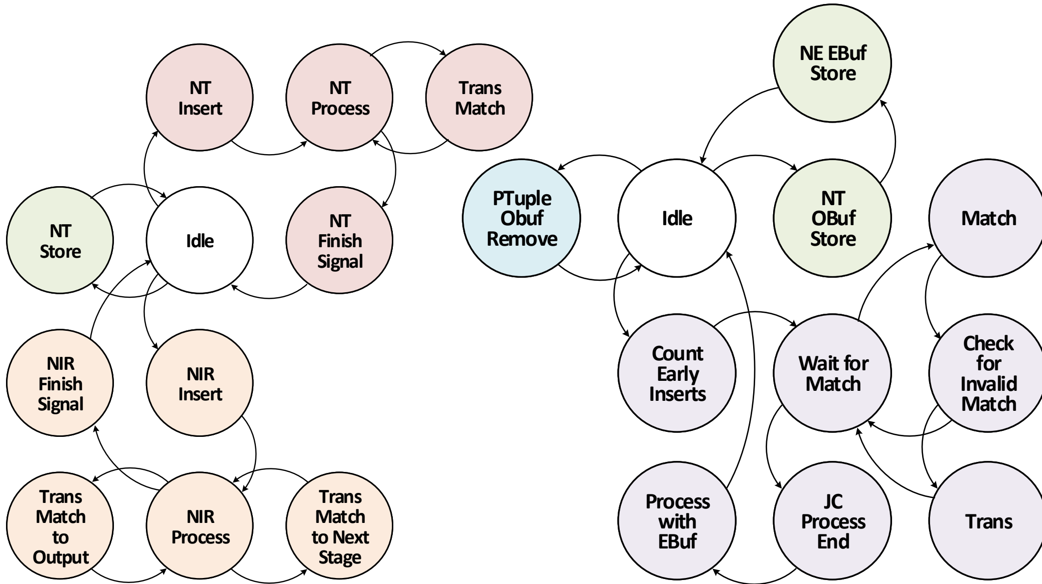


Figure 6.1.6: Join core lookup and store/expire controller state diagram.

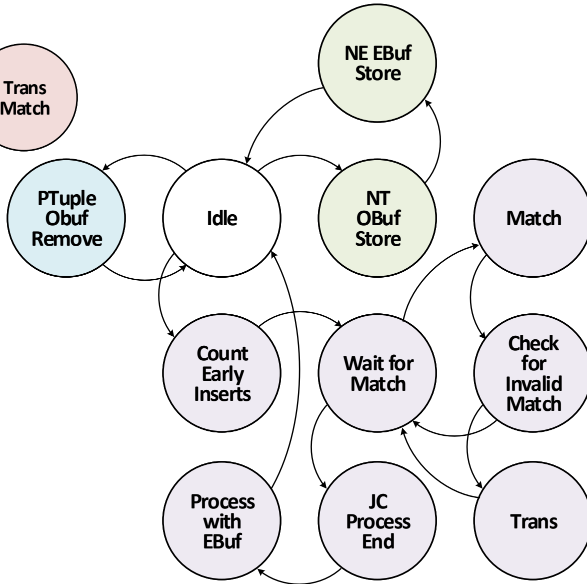


Figure 6.1.7: Order unit controller state diagram.

Figure 6.1.6 presents the state diagram for the join core lookup and store/expire controller. When receiving a new tuple belonging to this stage's sliding window from the distribution chain, the storage and expiration tasks are performed in the *NT store* state. When the new tuple belongs to the previous join core's window, it triggers the join execution. Here, the controller inserts this tuple into the join core in the *NT insert* state, while the actual processing occurs in the *NT process*. Whenever a match is found, it is sent to the next stage in the *trans match* state. The end of join execution is signaled to the next stage by sending an end flag in the *NT finish signal* state. Informing the end of execution is important since it enables the next stage to prepare to receive another set[4] of intermediate results or a new tuple.

After receiving an intermediate result, the join core controller feeds the relevant (extracted) tuple to the join core in the *NIR insert*. The processing occurs in the *NIR process* state, and the resulting data are either sent to another state as an intermediate result in the *trans match to next stage* or are sent to the result collection circuitry in the *trans match to output* state. The main difference in processing intermediate results in comparison to new tuples is that in the former, the controller waits for the end flag before it moves to the *NIR finish signal* state. This processing approach simplifies the refinement of results in the order control unit by avoiding the involvement of other new tuples in the middle of processing a set of intermediate results.

Order Unit Controller —

During processing, the order unit controller refines the resulting tuples to avoid any missed

[4]All the results produced in a join core from a new tuple or an intermediate result are referred to as a set.

or extra results. The order unit controller state diagram is shown in Figure 6.1.7. When a new tuple (received from the distribution chain) belongs to the current stage's window, in addition to being stored in the OBuf, it is also stored in the window and correspondingly expires a tuple that is stored in the EBuf. The storage of the new tuple in OBuf and EBuf is performed in the *NT OBuf store* and *NE EBuf store* states, respectively. The OBuf and EBuf both use a FIFO strategy. When the first tuple in the OBuf belongs to the current stage's window, the controller drops it from the OBuf and also drops the first (oldest) tuple in the EBuf.

After receiving a new tuple or an intermediate result to process, the order unit controller (using the OBuf) counts the number of previously stored tuples in the sliding window of the current stage in the *count early inserts* state. Note that the in-order insertion of new tuples in the OBuf preserves the actual tuple arrival ordering regardless of their origin. Subsequently, the order unit controller waits for the matches from the join core in the *wait for match* state. After receiving a match, the order unit controller transitions to the *match* state, and the match result is dropped if its constructing tuples exist in the prematurely stored tuples in the OBuf; otherwise, it is sent out to the next stage as an intermediate result in the *check for invalid match* and *trans* states, respectively.

After the end of processing in the join core, the state machine moves to the *JC process end* state, and the order unit controller processes the current tuple with the same number of tuples in the EBuf as the prematurely stored ones in the OBuf in the *process with EBuf* state. Finally, after receiving the end flag from the previous stage, the current tuple is looked up and removed from the OBuf in the *PTuple OBuf remove* state.

6.2 Multiway Stream Join with Stash

In the previous section, we focused on our scalable circular pipeline for multiway joins (Circular-MJ). In this section, we focus on optimizing the pipeline for nonindexable stream joins by introducing Stashed-MJ, which features novel buffering and improved pipelining. The first property of the Stashed-MJ design is the integration of a buffer, referred to as a *stash*, to materialize the intermediate results to substantially improve the throughput by avoiding recomputing previously processed tuples, as shown in Figure 6.2.2. The second property is centered around improving resource utilization. In Circular-MJ, only $N-1$ of N stages perform the processing such that, for each stream, one stage is always occupied with the storage and expiration tasks. Therefore, there is always one unused processing unit in the circular pipeline.

Integrating the stash into Circular-MJ (Figure 6.1.4) poses an important design challenge because we are now forced to share the buffer among multiple processing units, which are placed into two or more separate pipeline stages. This raises two concerns. First, having a shared block between

two pipeline stages that could be accessed frequently and in parallel violates the main concept of the circular pipeline design, which is based on the separation of concerns and a strictly one-way neighbor-to-neighbor communication. Second, this sharing can result in race conditions between the concurrent processing of a new tuple and storing the matched results of an earlier tuple, which now requires expensive pipeline stalls for coordination.

6.2.1 Stashed-MJ Design and Rationale

In Circular-MJ, we utilize three pipeline stages for a 3-way stream join operation. For every tuple insertion, two of these stages process this tuple against the sliding window of other streams, while the remaining stage is responsible for storing this tuple in its stream's sliding window and expiring the oldest tuple from it. There are two key insights that guide our new custom design: (1) storage and expiration operations are relatively less expensive compared to processing (particularly in nested-loop stream joins) and (2) storage and expiration are performed on a sliding window separate from the windows used for processing. By exploiting these insights, we reduce the number of pipeline stages to two by performing storage and expiration in parallel with the processing in the first pipeline stage. Consequently, the processing unit in the first stage has to operate on two sliding windows, depending on the newly received tuples' origin, but not simultaneously. This provides us with the opportunity to offload the processing operations of two streams (S_1 and S_2 in Figure 6.2.2) that are involved in updating the stash onto this stage, which eliminates the sharing challenge.

Stashed-MJ Design —

We present our Stashed-MJ architecture for a 3-way stream join including the stash (on the pair of streams S_1 and S_2) in Figure 6.2.1. The design is divided into three regions, each shown with a gray background. The upper region shows the main execution unit, including the two-stage pipeline, while the middle one depicts the interconnection circuitry, and the lower region specifies the memory components, consisting of sliding windows and the stash.

The first stage of the pipeline is responsible for the join operation on a new tuple against W_{S_1} (S_1 sliding window), W_{S_2}, or the stash. The actual execution is performed in the *processing* unit, and if there is an intermediate result, it is stored in the stash after passing through the *result storage* unit. The storage in the stash occurs after receiving a *grant* signal from the S_1&S_2 *storage and expiration* unit, indicating that the new tuple-related storage and expiration tasks are over. This results in better utilization of the stash due to the removal of expired intermediate results, which frees space for new ones, including the current tuple intermediate results. The second stage of the pipeline performs the join operation on a new tuple against W_{S_2} or W_{S_3}, in addition to the storage and expiration operations related to tuples from stream S_3.

The two-stage concept simplifies the design when considering the stash since the processing of

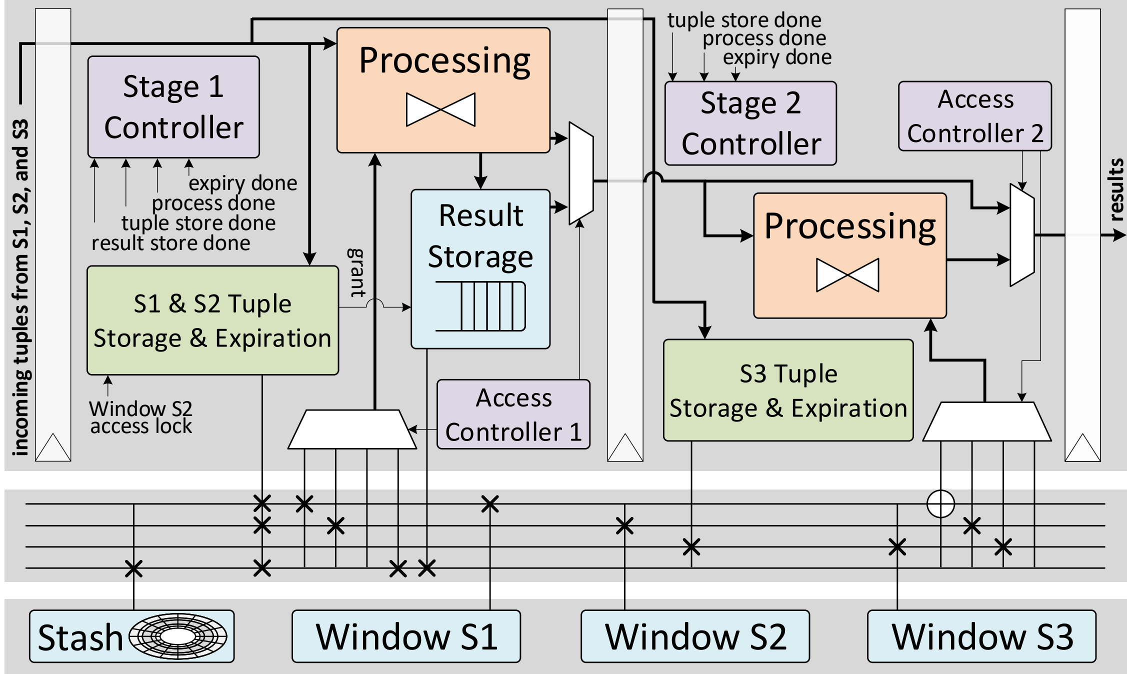


Figure 6.2.1: Stashed-MJ architecture.

tuples from streams S_1 and S_2 with W_{S_2} and W_{S_1}, respectively, is performed only in the first stage. Therefore, the intermediate results are produced on the same stage in which the stash is located. This eliminates transport of the intermediate results between stages and inherently removes their corresponding race conditions.

Stashed-MJ Operation —

After receiving a new tuple from stream S_1 (T_{S_1}), it is processed against W_{S_2} while being inserted into W_{S_1} and expiring the oldest tuple. The expired tuple's key is used by the S_1&S_2 *storage and expiration* unit to probe and expire all intermediate results in the stash that include this tuple. Meanwhile, the *result storage* unit enables the S_1&S_2 *storage and expiration* and *processing* units to operate in parallel, thus minimizing the time spent in this stage. While the intermediate results are being stored in the stash, they are also sent (one-by-one) to the second pipeline stage to be processed against W_{S_3}.

A new T_{S_3} is joined first against the stash only if it has all the intermediate results (no recent overwrite in the stash), and then the final results are emitted directly from the first stage. T_{S_3}-related store and expiration tasks are performed in the second stage by the S_3 *storage and expiration* unit. If the stash does not contain all of the intermediate results, then T_{S_3} is processed against W_{S_1} and W_{S_2} in the first and second stages, respectively.

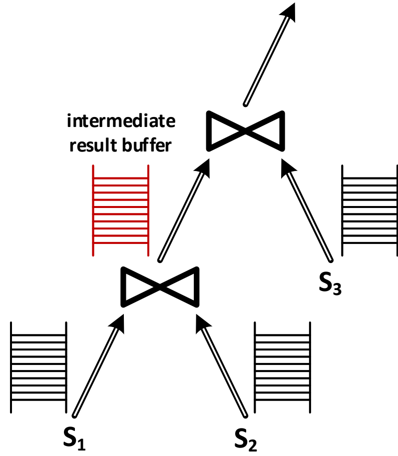


Figure 6.2.2: Intermediate result buffer.

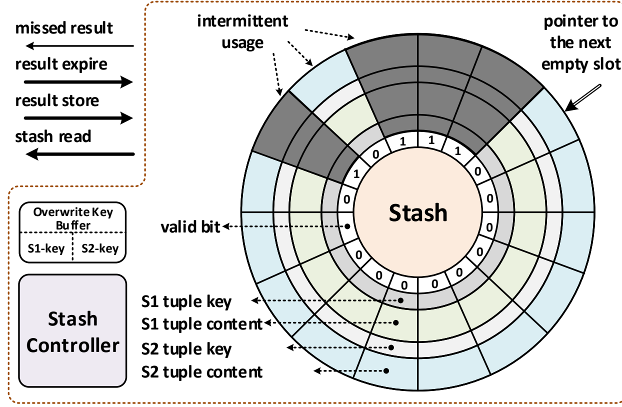


Figure 6.2.3: Stash design and its internal components.

Stash Effectiveness —

As demonstrated in Figure 6.2.2, materializing intermediate results in a buffer accelerates the entire execution for tuples from stream S_3 since they are only compared with the intermediate results in this buffer. However, there are two major challenges regarding this buffer, which we elaborate on next.

The first challenge is the buffer's effectiveness with respect to the processing throughput. Assume that we have a stash for a pair of streams S_1 and S_2 similar to Figure 6.2.2. Receiving tuples from streams S_1 and S_2 does not benefit from this buffer since they have to be processed against W_{S_2} and W_{S_1}, respectively, and then against W_{S_3}. However, a tuple from stream S_3 only needs to be processed against the materialized intermediate results, whereas without this buffer, we have to reorder the join operators, process T_{S_3} with W_{S_1} and then process the intermediate results within W_{S_2}. Furthermore, this buffer provides a tradeoff between using increased storage vs. reduced computation, which can improve system power consumption in addition to enhancing the processing throughput when properly utilized.

Depending on the match probability, the number of intermediate results could vary from a few tuples up to a size of $W_{S_1} \times W_{S_2}$, which makes the size of this buffer another concern. Opting for a small size increases the possibility of overwrite, which renders the existence of this buffer ineffective since we need to reprocess tuples from stream S_3 with both W_{S_1} and W_{S_2} to avoid missing any results. Meanwhile, a large buffer could make the solution less effective by consuming a large amount of memory.

To address these challenges, the intermediate buffer can be effective if the following two conditions hold: (1) the expected tuple rate from stream S_3 is high compared to other streams since this buffer

is only used when receiving tuples from this stream and (2) the expected number of intermediate results is not great compared to the sliding windows' size, which is further elaborated in Section 6.2.3.

6.2.2 Stash Internal Architecture and Operation

We use a circular buffer that stores intermediate results (any pair of tuples that have satisfied the join condition) in addition to a *valid bit* for each. A true (logical value of one) valid bit shows that the content for that buffer slot is valid and vice versa. Figure 6.2.3 shows the main components for the stash.

Stash Expiration —

Upon receiving a new tuple, the $S_1\&S_2$ *storage and expiration* unit, shown in Figure 6.2.1, stores the tuple in its corresponding sliding window. At the same time, this unit also expires the oldest tuple in the sliding window, which is followed by an expiration process on the stash. The $S_1\&S_2$ *storage and expiration* unit uses the key of the recently expired tuple to search for the intermediate results that include this key in the stash. Expiration in the stash is performed by setting the valid bit of the expired intermediate results to false.

The expiration process can expire the intermediate results from any location in the stash. This leads to a noncontinuous use of the stash, which we refer to as *intermittent usage.* The intermittent usage forces a full stash search for every probing to avoid missing any results.

Stash Insertion —

Processing new tuples in the *processing* unit produces intermediate results that are initially stored in the *result storage* unit. After receiving the *grant* (ready) indication from the $S_1\&S_2$ *storage and expiration* unit, which indicates the end of insertion in the sliding window and expiration from the stash, the *result storage* unit starts storing its intermediate results in their specified (by the stash controller) locations and correspondingly sets their valid bits to true. After each storage, the *stash controller* searches for another empty location for the next intermediate result insertion.

Invalid Stash —

Based on the join selectivity, the number of intermediate results could exceed the size of the stash, which leads to overwriting of the (still valid) intermediate results. We refer to this state of the stash as *invalid* since after overwriting, the stash no longer

contains all intermediate results, which makes it unusable. In this invalid state, the pipeline design in Figure 6.2.1 performs its normal operation while ignoring the intermediate results in the

stash. However, the stash continues its storage and expiration operations with the exception that it only stores the new intermediate results in a continuous order without checking whether each slot is empty. This makes it possible for the stash to recover from the invalid state.

Stash Recovery —

For each overwrite, the *stash controller* preserves the latest keys from both tuples (in the intermediate result) in the *overwrite key buffer*. After every expiration, the controller checks the expired tuple's key with each of these preserved keys in the *overwrite key buffer*; when there is a match, the controller marks that key as *passed*. If both of the preserved keys receive a *passed* mark, then the stash returns to its valid state.

As an example of the stash recovery, assume that Tx_{S_1} generates the last intermediate result (Tx_{S_1}-Ty_{S_2}), which causes an overwrite in the stash. Therefore, the keys of these two tuples are preserved in the *overwrite key buffer*. Receiving an expiration request for Tx_{S_1} means that there has been no overwrite on the intermediate results for tuples in W_{S_1} after the insertion of Tx_{S_1} since Tx_{S_1} has performed the last overwrite itself. The same concept also applies to Ty_{S_2} after receiving its expiration. Again, this means that there has been no overwriting of the intermediate results for tuples in W_{S_2} after the insertion of Ty_{S_2}. Summarizing the last two statements implies that the stash currently contains all intermediate results.

We refer to this transition to the valid state as *stash recovery*. Note that a new overwrite again resets both of the preserved keys' statuses to the *failed* mark. Additionally, shifting the intermediate results to remove the gap created by the expiration can improve the recovery process. This eliminates the situation in which we overwrite a valid intermediate result when there are still empty locations in the stash.

6.2.3 Stashed-MJ Analysis

Here, we present an analytical evaluation of Stashed-MJ with both disabled and enabled stashes.

Stashed-MJ with Disabled Stash —

The processing latency is calculated as follows:

$$\begin{aligned} \text{Latency} = & P(T_{S_1}) \times (T_{proc(W_{S_2})} + mr_{(T_{S_1},W_{S_2})} \times T_{proc(W_{S_3})}) + \\ & P(T_{S_2}) \times (T_{proc(W_{S_1})} + mr_{(T_{S_2},W_{S_1})} \times T_{proc(W_{S_3})}) + \\ & P(T_{S_3}) \times (T_{proc(W_{S_1})} + mr_{(T_{S_3},W_{S_1})} \times T_{proc(W_{S_2})}) \end{aligned} \quad (6.2.1)$$

where $P(T_{S_x})$ defines the probability of having a tuple from stream S_x; match-rate ($mr_{(T_{S_x},W_{S_y})}$)

specifies the average number of resulting tuples per inserted new tuple (T_{S_x}), which is determined by the match probability and the size of the sliding window (W_{S_y}) used for comparison; and $T_{proc(W_{S_x})}$ determines the required time to process a specific sliding window (W_{S_x}) against a tuple.

Note that we no longer have T_{store_expire} in the equations since the storage and expiration operations are overlapped with the processing operation on another window.

If we assume that all $P(T_{S_x})$, $T_{proc(W_{S_x})}$, and $mr_{(T_{S_x},W_i)}$ are identical, then the equation simplifies to:

$$\text{Latency} = T_{proc(W)} \times (1 + mr_{(T,W)}) \tag{6.2.2}$$

For the throughput evaluation, we observe that the time to process one tuple in the second stage of the pipeline is interleaved with the time to process the next incoming tuple in the first stage since they are executed in parallel. This pipelining effect also applies to other stages, leading to the following throughput equation for our two-stage pipeline, assuming similar properties for all streams:

$$\text{Throughput} = \frac{1}{T_{proc(W)} \times (1 + U(mr - 1) \times (mr - 1))} \tag{6.2.3}$$

where $U(x)$ is the *unit step function*. This equation also confirms that for `match_rate` ≤ 1, the throughput of the system remains independent of the number of streams and correspondingly the number of pipeline stages.

Stashed-MJ with Enabled Stash —

The inclusion of the stash in the pipeline architecture changes the time required to process a tuple for stream S_3 compared to the previous equations. Assuming that we have a stash in the valid state, the average processing latency for a tuple that uses this stash is calculated as follows:

$$\text{Latency for } \mathrm{T}_{\mathrm{S}_3} = T_{proc(Stash)} \tag{6.2.4}$$

since T_{S_3} is only compared against the stash in the first stage and the results are emitted directly from this stage. Note that the time to store and expire in/from W_{S_3} is a small constant and is overlapped by the processing of T_{S_3} in the first stage.

For tuples from S_1 and S_2 that do not benefit from the stash, the latency is calculated using the previous equations. Although the match rate does not exist in this equation, it and the size of the sliding windows define the proper size for the stash that affects the latency. Since we only have one working stage in our pipeline when using the stash, the throughput would be the reverse of

latency for the stream that uses the stash (here, S_3).

$$\text{Throughput for } \mathrm{T}_{\mathrm{S}_3} = \frac{1}{T_{proc(Stash)}} \tag{6.2.5}$$

Using the evaluations for the throughput of Stashed-MJ with and without the stash, the effectiveness of the stash is analytically confirmed if the following condition holds:

$$T_{proc(Stash)} \leq T_{proc(W)} \times (1 + U(mr-1) \times (mr-1)) \tag{6.2.6}$$

The processing time is directly related to the number of tuples for the comparison. Consequently, using a *stash* $\geq$ *sliding window* not only provides no improvements but also imposes performance penalties due to the additional circuitries and memory resources.

6.2.4 Parallel Multiway Stream Join

The stream join parallelization model (presented as SplitJoin [68]) relies upon a single data flow (uni-flow) that is shared between streams. Therefore, there is no need to have a per-stream-dedicated data path, and we are not limited by the number of streams. This allows us to hide the multiway stream join implementation internally within each global[5] join core (G-JC) without influencing the overall parallel architecture.

To integrate our Circular-MJ (or Stashed-MJ but with disabled stash) into the uni-flow model, we use the same architecture presented in Figure 6.2.1 with the addition of *store-expire turn* counters in the input of each Circular-MJ which resides inside a G-JC. In the parallel architecture, referred to as Parallel-MJ, we replicate the G-JCs to obtain linear throughput scaling.

Store-expire turn counters preserve the position of a G-JC among other G-JCs (P), which is later used to calculate the storage turn of incoming tuples in parallel stream joins based on the uni-flow model. Each counter (C_{S_j}) is dedicated to a stream (S_j) and counts the number of tuples received from that stream. In each G-JC (i), a new tuple from (S_j) is stored, and the oldest tuple is expired from its corresponding sliding window when ($i \equiv C_{S_j} \mod P$), while all other global join cores process this tuple in parallel. Except for the storage and expiration operations, which are performed in a round-robin fashion, the Circular-MJ pipeline remains as before, and it executes the join operators on each incoming tuple in parallel with others in other G-JCs.

[5] We refer to each separate compute unit in the parallelized architecture as a global join core (G-JC). Each G-JC can contain a Circular-MJ (or a Stashed-MJ with disabled stash) realization.

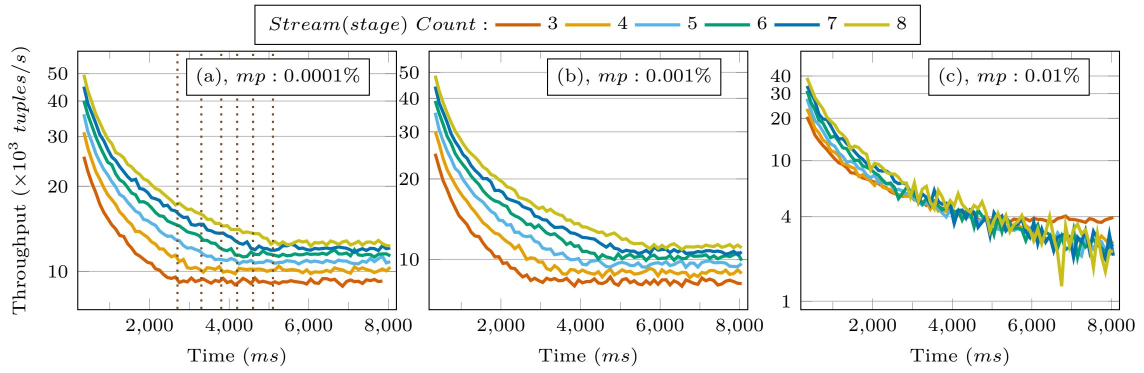


Figure 6.2.4: Stream count effect on input throughput ($w : 2^{14}$).

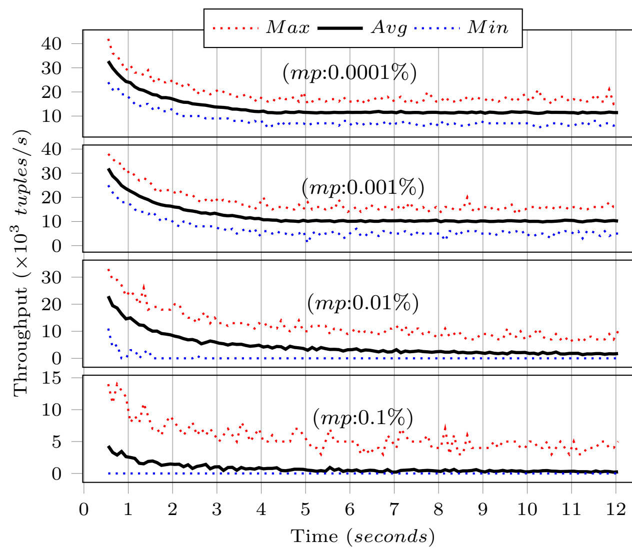


Figure 6.2.5: Match probability effect on throughput ($6 - way, w : 2^{14}$).

6.3 Stashed-MJ Extensions & Discussions

6.3.1 Stashed-MJ Window Access Pattern

In the two-state pipeline since W_{S_2} is shared between the first and second stages, in a few cases it is locked by the second stage to avoid any race condition between the processing of tuple in the second stage and storage of a newer tuple in the first stage.

Table 6.3.1 shows the access pattern in each stage of the pipeline (Figure 6.2.1) while receiving new tuples. Here *PU* refers to the *processing* unit, and *SEU* refers to the S_1&S_2 *storage and expiration* unit. Since we have two stages and three streams, we have nine arrival order (tuples' origin) possibilities. In this table, we observe a read (T_{S1} & T_{S3}) and a write (T_{S2} & T_{S3}) access conflict, which are shown by green and red colors, respectively. The read access conflict can be resolved by using a dual read port memory for W_{S_2}, but we need to stall the pipeline to address the write access conflict, which is handled by the *window* S_2 *access lock* signal in Figure 6.2.1. Having the stash (in the functional state) eliminates both of these conflicts since all processing for T_{S3} is performed in the first pipeline stage.

6.3.2 Stash Storage Pattern in the Invalid State

As discussed earlier in Figure 6.5.9, at approximately 450ms we observe a peak that brings the stash to the invalid state. The detailed behavior of the stash in this state is shown using the valid bits in Figure 6.3.1. The first signal (from the top) in this diagram shows the status of the stash, i.e., valid (normal operating) or invalid. After the overwrite, this signal is pulled up, and the stash goes into the invalid state while monitoring expirations for recovery. Here, we observe how the storage pattern changes from almost random (based on available slots freed by previous expirations)

Table 6.3.1: Pipeline stages access pattern (disabled stash).

Pipeline Stage 1			Pipeline Stage 2		
Tuple	PU	SEU	Tuple	PU	SEU
T_{S1}	W_{S2}	W_{S1}	T_{S1}	W_{S3}	–
T_{S1}	W_{S2}	W_{S1}	T_{S2}	W_{S3}	–
T_{S1}	W_{S2}	W_{S1}	T_{S3}	W_{S2}	W_{S3}
T_{S2}	W_{S1}	W_{S2}	T_{S1}	W_{S3}	–
T_{S2}	W_{S1}	W_{S2}	T_{S2}	W_{S3}	–
T_{S2}	W_{S1}	W_{S2}	T_{S3}	W_{S2}	W_{S3}
T_{S3}	W_{S1}	–	T_{S1}	W_{S3}	–
T_{S3}	W_{S1}	–	T_{S2}	W_{S3}	–
T_{S3}	W_{S1}	–	T_{S3}	W_{S2}	W_{S3}

to contiguous, as the stash controller sequentially stores intermediate results regardless of the valid bits' status. This change in pattern simplifies the recovery circuitries. After recovery, the status signal returns to zero (valid state), and the storage pattern returns to its previous form.

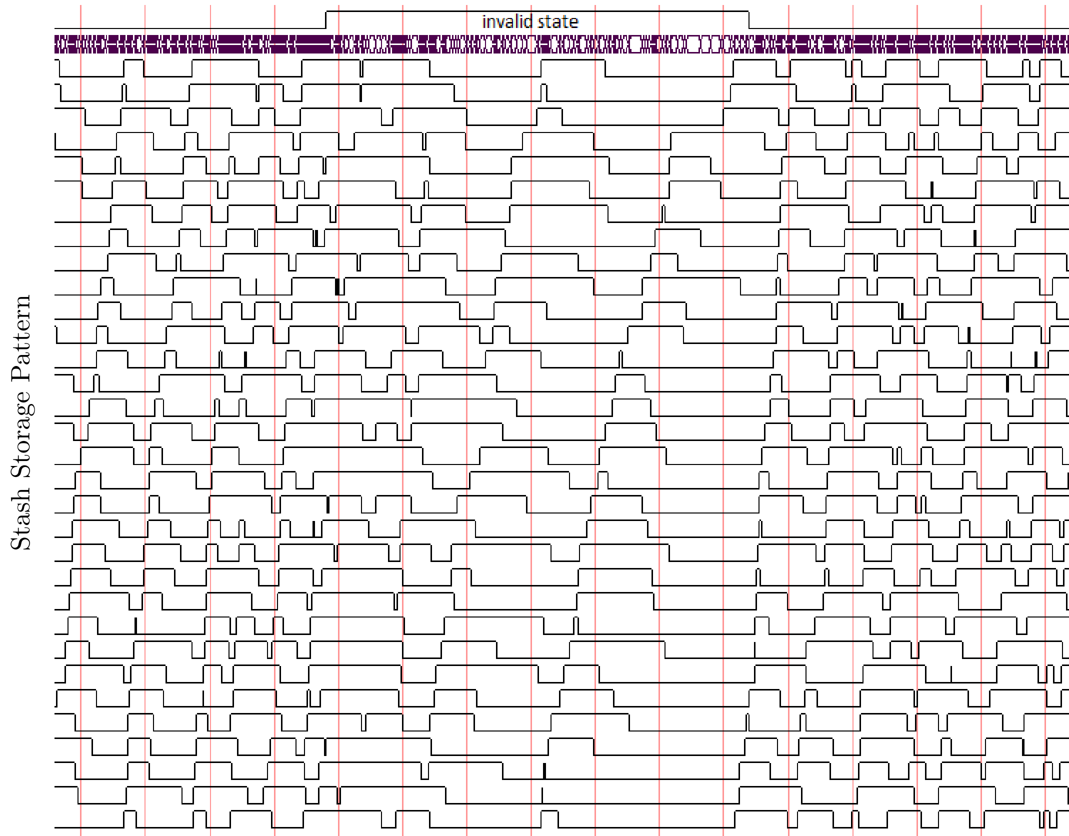


Figure 6.3.1: Stash behavior in the invalid state ($w : 2^{10}, s : 2^5, mp : 0.001\%$).

6.3.3 Stash with Recomputation Support

One challenge with the original stash design is its limited functionality when it is in the invalid state; depending on the stash size, sliding window size, and match rate, there could be a scenarios in which the stash remains mostly in the invalid state by just a few extra overwrites compared to its size. Another challenge arising from the intermittent usage is the necessity to process the whole stash regardless of the number of intermediate results in it.

To address these challenges, we propose an extension to our stash design to support partial recomputation when the stash is in the invalid state, which also eliminates the intermittent usage and enhances stash processing throughput. Figure 6.3.2 presents our enhanced stash design with an extra field of the sliding window index for each storage location and an *Overwrite Index Buffer*

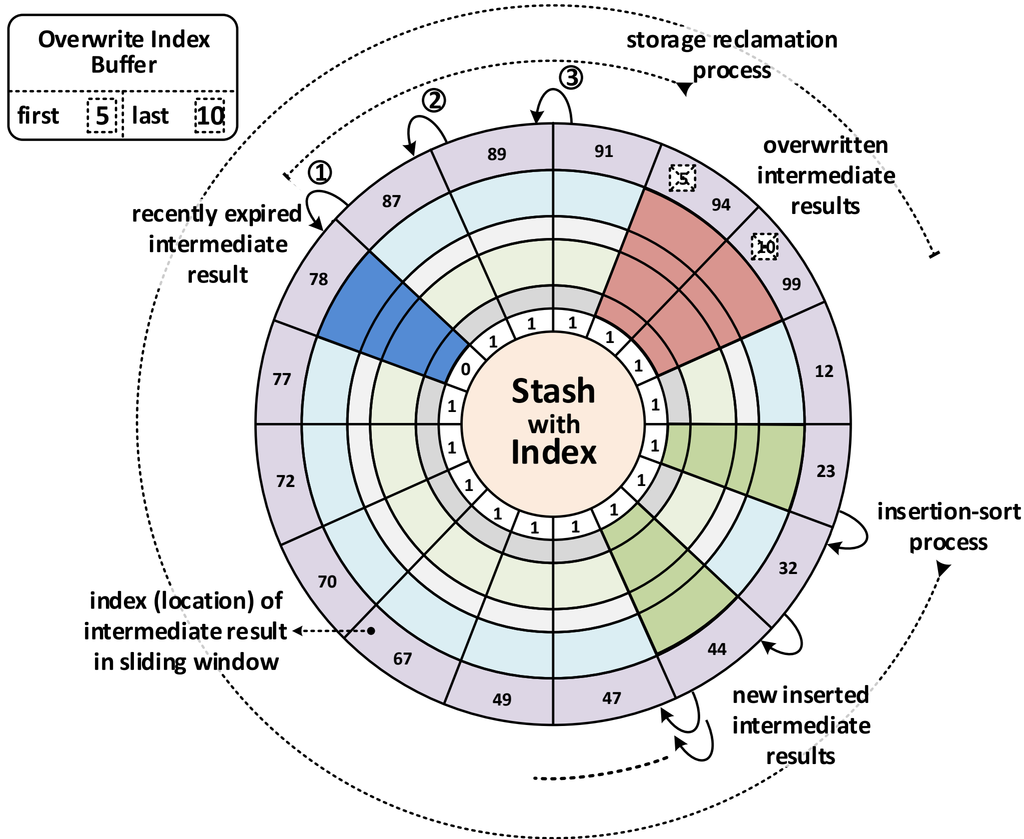


Figure 6.3.2: Stash with recomputation support.

in addition to the existing components in the original design.

Addressing Intermittent Usage —

In our enhanced stash, after expiration, we have the *storage reclamation* process, which shifts the intermediate results to omit the gaps caused by the expiration. As a result, after this process, all intermediate results are placed consecutively in the enhanced stash without a gap. The procedure of omitting gaps by the storage reclamation process is shown in Figure 6.3.2, where the recently expired result is indicated by blue color. The important benefit of this is limiting stash computation only to the available intermediate results and not the whole stash. In the worst-case scenario, the storage reclamation process needs to traverse (shift results) across the whole stash to omit gaps created by the expiration; however, there is no penalty since the reclamation process overlaps with the processing of the current tuple with its corresponding sliding window.

Partial Recomputation —

Every time the stash stores an intermediate result, it also saves the sliding window index of that result in the same location (outer ring in Figure 6.3.2). When the stash is full, the next intermediate result overwrites an existing one, and the index of the overwritten result is recorded in the *first* field of the *Overwrite Index Buffer* table. Afterward, every new overwrite in the stash updates the *last* field of this table with the index of the overwritten result.

The enhanced stash uses an *insertion-sort process* to keep the intermediate results sorted in relation to their sliding window index. Suppose the stash has entered the invalid state; the processing unit uses the fields in the *Overwrite Index Buffer* table for recomputation of the removed (overwritten) intermediate results instead of processing the whole sliding window. For example, the recomputation range for the stash in Figure 6.3.2 is from five to ten. Our pipeline solution considers sliding windows as circular buffers; as a result, having a smaller value in the *last* compared to the *first* field implies that the interval to recompute is from the *first* index to the end of the window and then from the beginning of the window to the *last* index.

Insertion-Sort Process —

The key factor that enables the partial recomputation is the *insertion-sort process*, which runs after the storage reclamation and stores the new intermediate results in their respective slot in the stash. To find the right slot, the stash controller starts from the element with the minimum index value and traverses the stash until it reaches an element with an index bigger than the index of the current intermediate result. It then shifts other elements in the direction toward larger indexes to open a free slot.

The actual sliding window execution starts from an index of zero to the end of the sliding window, and the intermediate results are produced in increasing order with respect to their sliding window index. Thus, while processing the current tuple, for every extra new intermediate result, the insertion-sort process resumes searching from the slot next to the previously inserted intermediate result. By choosing a reasonable size (determined based on the expected number of intermediate results and with an upper bound of sliding window size) for the stash, there are only limited penalties since all auxiliary operations are performed in parallel with the processing of the current tuple with its corresponding sliding window. When the stash is full, the insertion-sort process leads to overwriting of the intermediate results with lower index values as shown in Figure 6.3.2. Here, two new insertions (shown in green) lead to two overwrites (shown in red), which are later handled by the partial recomputation concept.

Notably, by utilizing the new stash, it is no longer possible to merge intermediate results of S_1-S_2 and S_2-S_1 into one stash because the enhanced stash is sorted based on sliding window indexes. Therefore, we need separate stashes for S_1-S_2 and S_2-S_1.

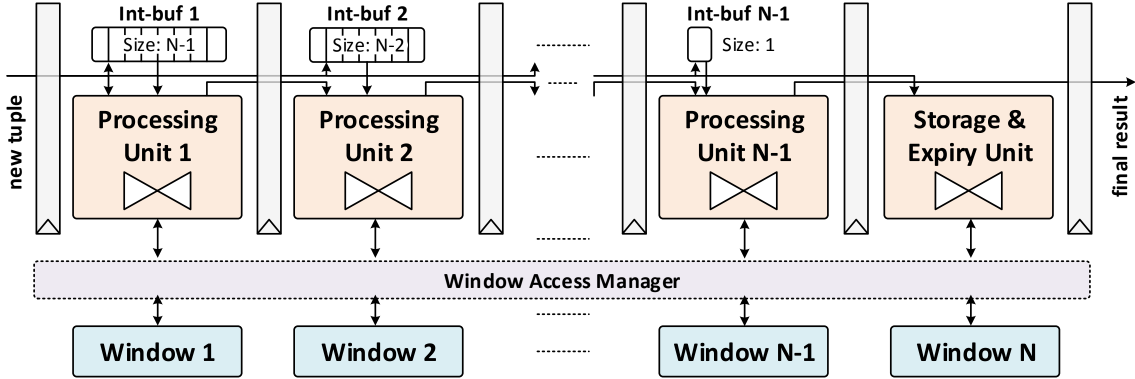


Figure 6.4.1: Linear-MJ architecture.

6.4 Supplemental Material

6.4.1 Multiway Stream Join Architectural Challenges

Using the right-deep join tree architecture, we reach a set of linearly connected processing stages where each one performs a join operation relevant to the current tuple under processing; correspondingly, the intermediate results are sent to the next stage for further processing. We propose a linear pipeline multiway stream join architecture (Linear-MJ) based on this arrangement, shown in Figure 6.4.1, to further describe the challenges of a straightforward hardware architecture.

In this architecture, storage and expiration tasks are challenging since each processing unit (in each stage) has the possibility to access each one of the sliding windows. Storing a tuple in advance (i.e., in the first stage) may lead to missing results since the oldest tuple will expire sooner than its order. Similarly, storing a tuple later (i.e., in the last stage) in the pipeline delays expiration of the oldest tuple from its corresponding window, which may lead to invalid results from the match between newer tuples and the oldest tuple.

To deal with this issue, we use a buffer (*Int-buf*) in each pipeline stage starting with a size of $N-1$, assuming we have N streams, from the left-most stage to the size of 1 in the last processing stage. In the last stage of the pipeline, we use a dedicated storage & expiry unit to store new tuples and expire oldest ones. Each new tuple overwrites the oldest tuple on Int-Buf, while a copy is sent to the Int-Buf of the next stage when its processing in the current stage is over. Since we delay the storage of new tuples to the last pipeline stage, each stage takes care of in-order processing to guarantee correct results. In each stage, all new tuples that arrived prior to the current tuple, which are under processing and belong to the sliding window in this stage (assume m), are also considered for processing. Consequently, we need to exclude the matches that are from the last m tuples in the sliding window since they were supposed to be expired.

Architecture Drawbacks —

Although Linear-MJ utilizes point-to-point communications between the processing units, it lacks the scalability perspective due to the $N-to-N$ crossbar switch present in the window access manager of Figure 6.4.1. The existence of this switch is necessary for this design since each pipeline stage can implement any join operator, and therefore it can access any of sliding windows. Another drawback of the Linear-MJ architecture is the window access conflicts that may arise from the concurrent read requests to the same window from multiple join cores, more severe concurrent read requests (from join cores), and write requests from the storage and expiry unit. Although the multiple read requests can be addressed by optimizing the crossbar switch and also using locks to mitigate the concurrent read-write request conflicts, these optimizations and locks put a great strain on the design complexity, resource consumption, and actual processing performance of the hardware realization.

6.4.2 Effects of Parallelization on Match Burst Rate

One challenge for the stash in our multi-stream join is when there are large bursts of tuples that satisfy the join operator conditions. One of the interesting features of parallelization using flow-related algorithms (i.e., uni-flow) is that they inherently mitigate this issue since they distribute tuples (usually in a round-robin fashion) among independent join cores. As a result, each join core receives only a fraction of the bursts, and this fraction decreases as we increase the number of (global) join cores.

6.4.3 Time-Based Multiway Stream Join

Extension of the pipeline multiway stream join and stash architecture for the time-based sliding window is rather straightforward by updating the expiration logics. In the time-based sliding window, we preserve the timestamps that are later used for expiration; therefore, the expiration can be postponed to the time of processing a sliding window or a stash. The rest of the architecture remains the same as the current solution for the count-based sliding windows.

6.4.4 Extending the Buffering (Stash) for All Streams

Generalizing the concept of the stash poses several fundamental challenges when going beyond three streams in a pipeline architecture.

In our Stashed-MJ design, the (first) stash stores the intermediate results for the processing

of stream S_1 against stream S_2 and vice versa. Therefore, there is a significant speedup in the processing of the tuples from stream S_3 since they are only considered against the intermediate results in the stash instead of being compared with the whole sliding windows of streams S_1 and S_2. Note that the stash concept utilization is rewarding when the following condition is expected: $stashsize << slidingwindows'sizes$ (S_1 and S_2 streams). The application of the stash concept on the other two streams proves to be challenging, which we explain in the following descriptive example.

Assume we add a second stash for another pair of streams, e.g., S_1 and S_3. As a result, we expect to have a speedup in the processing of tuples from stream S_2 since they are only considered against the stash on streams S_1 and S_3. However, this is not the case in practice. Indeed, there is a speedup in the processing of tuples from stream S_2 against their corresponding stash, but we still need to process these tuples against sliding window S_1 to produce intermediate results for the first stash on streams S_1 and S_2.

Therefore, although tuples from stream S_2 are processed faster, the pipeline still has to wait for the processing of these tuples with sliding window S_1 to produce intermediate results for the first stash (on streams S_1 and S_2). Note that this was not the case with the first stash since there was no further processing of tuples from stream S_3 against the other streams' sliding windows. Thus, incorporating additional stashes becomes ineffective. In this regard, the proposed concept is effective for the use cases where a stream (that benefits from the stash) has a significantly higher input rate than the other streams, as evaluated in Figure 6.5.12b.

6.4.5 Hardware Optimization Using a Stash

Stash size is expected to be small (assume that this is a condition) compared to the sliding window size. Hardware platforms commonly offer some small portions of internal memory (like BRAMs in FPGAs) that provide huge bandwidth. Assuming the stash condition size holds, we can benefit from this fast internal memory for stash realization. Additionally, we can process multiple intermediate results (stored in the stash) in parallel due to the large access bandwidth of these memories. This provides a linear speedup in relation to the number of intermediate results processed in parallel.

6.4.6 Extending Stash for Parallel-MJ

Incorporating the stash concept inside each G-JC proves challenging, which we explain using an example. Assume we have a 3-way stream join, where each stream's sliding window is divided (equally) among G-JCs in Parallel-MJ. In this example, we also assume that our Parallel-MJ has 6 G-JCs. This way we will have 6 instances of Stashed-MJ and each one resides inside one of the G-JCs.

Note that each Stashed-MJ keeps only one-sixth (referred to as a sub-window) of the sliding window for each stream and correspondingly stores one-sixth of the received tuples from each stream.

Now, we receive a Tx_{S_1}, which is stored in its corresponding sub-window in the first G-JC. Assume Tx_{S_1} also produces intermediate results (Tx_{S_1}-Ty_{S_2}) and (Tx_{S_1}-Tz_{S_2}) in the first and second G-JCs, respectively. These two intermediate results are also stored in the stash of their corresponding G-JCs.

After a period, Tx_{S_1} is expired from the sub-window in the first G-JC. Correspondingly, its intermediate result (Tx_{S_1}-Ty_{S_2}) is also expired from the stash in the first G-JC. However, (Tx_{S_1}-Tz_{S_2}) remains in the stash of the second G-JC since it has not seen the expiration of Tx_{S_1}, which occurred in the first G-JC. This intermediate result can produce incorrect final results since it is processed by upcoming new tuples. Therefore, utilizing stash in the parallelized solution and in a scalable manner, namely, avoiding central coordination among G-JCs, remains a challenge to be addressed in future works.

6.4.7 Stage Buffer Potentials in Circular-MJ

As a side benefit, the existence of a stage buffer improves pipeline performance by eliminating the time that a stage needs to wait for the next stage to become ready to receive the next intermediate result. If a join condition (in a stage) is expected to have low selectivity, it is possible to selectively increase the buffer size of the next pipeline stage instead of over-provisioning buffers' sizes for the whole pipeline. Additionally, having multiple intermediate results ready at the entrance of a join core opens the opportunity to process them concurrently. This is done by swiping a sliding window once while comparing it with multiple intermediate results (instead of one) in the nested-loop algorithm to boost the processing throughput, which we leave for a future work.

6.4.8 Ensuring Match Result Correctness Using the Control Unit

The order control unit (Figure 6.1.5b) uses the preserved arrival order in OBuf and EBuf to avoid missing matches. Consider the specified data for OBuf and EBuf components in Figure 6.4.2 and a join condition between streams S_2 and S_3. In this snapshot, both tuples from stream S_3 have arrived after tuples from streams S_2 and S_1, but they are stored in the sliding window and have expired at $T_{(n)S3}$ and $T_{(n+1)S3}$. When the next tuple to process is an intermediate result that contains T_{S2}, the refine unit (Figure 6.1.5b) drops the matches between this intermediate result and $T_{(x)S3}$ and $T_{(x+1)S3}$ while the compare unit looks for the matches between this intermediate result and $T_{(n)S3}$ and $T_{(n+1)S3}$. The number of tuples to skip from the head of sliding window, which is also the number of tuples to consider from EBuf, is calculated by counting the tuples that

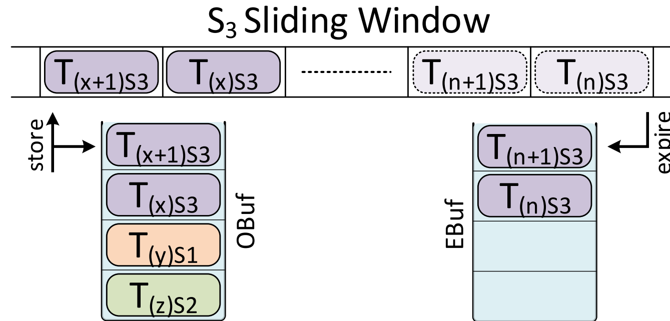


Figure 6.4.2: Order control unit operation example.

have arrived after the tuple under processing (here the intermediate result contains T_{S2}) and is stored in the sliding window).

6.4.9 Rationale to Fix the Join Order as the Right-Deep Join Tree

To ensure flexibility in changing the order of join operators on the fly, we need to have a broadcast connection from each join core to all other join cores. For example, with 6 streams, we need to have 6×(256-bit or more, depending on the size of the final results of the multiway join operator) outward connections between each join core and the other join cores (this translates to at least 6 large multiplexers and 6×6×256-bit wires), even without considering the insertion and result gathering circuitries. Unlike in software solutions, such large connections are not suitable for a hardware realization, at least not for general cases. In fact, we argue that even when a realization is possible, its performance and scalability characteristics are questionable.

In software, connections like these are readily (and almost trivially) possible through the shared memory hierarchy without significant penalties since there are no physical wirings between the operators. In that case, we need to transfer an intermediate result to one of the join core routines (implemented in software); the corresponding routine simply reads this intermediate result, which is stored in the main memory. Even with proper optimizations, the intermediate result can be also kept in the processor's cache (or even the processor's internal register banks), which could reduce the data access latency significantly.

For the algorithms used in software, it is not advisable to fix the join orders as the right-deep tree since we need to reorder the join operators on the fly based on the currently received tuple. However, our approach benefits from a circular design that incorporates a fixed-order right-deep join tree to avoid the large number of connections and multiplexers between the join cores to be able to implement our solution in hardware.

6.5 Experimental Results

We realized our scalable multiway stream join pipeline (Circular-MJ), optimized pipeline with stash (Stashed-MJ), and parallel pipeline (Parallel-MJ) in VHDL, and we synthesized and implemented it on a Virtex-UltraScale XCVU190 FPGA. For evaluations, we used Questa Advanced Simulator to extract cycle-accurate measurements (with a clock frequency of 100 MHz), thus guaranteeing the same performance for the actual hardware. For the synthesis and implementation steps on the FPGA, we used the Xilinx Vivado 2017.2.1 tool, and our VHDL realizations are also valid for building an ASIC solution.

Each input stream consists of 64-bit (32-bit key and 32-bit value) tuples that are joined against other streams' sliding windows. Our realizations have the ability to adopt larger tuples that are defined by presynthesis parameters. In the experimental results, w indicates the window size, and mp stands for the match probability.

We use the same probabilities to have a new tuple from each individual stream, unless explicitly specified. We utilize both uniform and normal distributions[6] and a range limiter function[7] from the OSVVM library[8] to assign a stream identifier (origin) for each new tuple. We have demonstrated two examples for the stream origin assignment using μ:2, σ:1 and μ:5, σ:2 parameters in Figure 6.5.4.

6.5.1 Circular-MJ Evaluations

In the first part of our evaluations, we consider the effect of different parameters on our Circular-MJ realization. Our focus here is on the design and architecture of the pipeline, which is orthogonal to the choice of join algorithms employed. In our experiments, we primarily use a nested loop in each join operator. This allows better evaluation of our pipeline properties since the long processing times for each tuple force intermediate results to accumulate in the stage buffers and put the pipeline under stress. Additionally, we present throughput measurements for a Circular-MJ using a hash-based join to demonstrate the independence of our pipeline designs from the choice of join algorithms.

Stream Count Effect —

To measure the effect of the number of streams on the input throughput, we use join conditions with high selectivity (low match probability) to approach the maximum sustainable throughput, presented in Figure 6.2.4a. As expected due to the parallelism resulting from pipelining, an increase in the number of pipeline stages linearly improves the processing throughput. This shows the

[6] RandTyp := Op.setRandomParm(NORMAL, Mean, Deviation).
[7] RandVal := Op.Randslv(1, 6).
[8] The Open Source VHDL Verification Methodology.

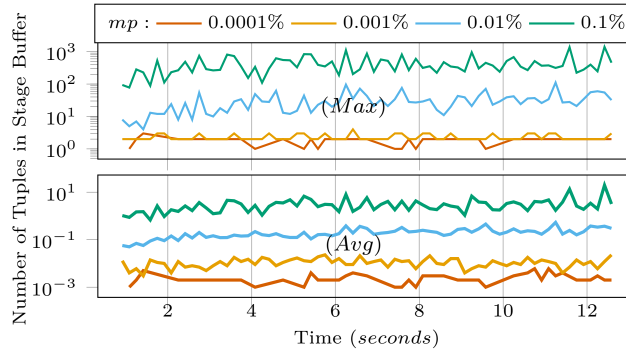


Figure 6.5.1: Match probability effect on stage buffer usage ($1^{st} stage, 6-way, w : 2^{14}$).

effectiveness of the distribution chain since it is able to keep the pipeline stages busy, which is the key factor in achieving parallelism via pipelining. On the left side of this figure, we first observe a warm-up phase where we have a super-linear reduction in the input throughput as the sliding windows are filled. The vertically dash lines specify the end of the warm-up phase for pipelines with a different number of streams.

Figures 6.2.4b and 6.2.4c present the input throughput for lower selectivities. We only observe a small reduction in the input throughput in Figure 6.2.4b, while the measurements for match probability (mp) of 0.01%, Figure 6.2.4c, show much lower and also sporadic throughput readings due to overloading and congestion of the pipeline stages. Due to the higher match probability, we have more intermediate results at each pipeline stage, which is the reason for the reduced throughput. Here, each pipeline stage receives many intermediate results from its previous stage, which prolongs the reception of new tuples. Therefore, the throughput, which is the number of new tuples inserted per second, drops to approximately 1000 $tuples/s$ (note that y-axis values should be multiplied by 10^3). Because there are more streams, a larger number of intermediate results are produced (e.g., per 8 streams instead of per 5 streams), which has a greater impact on the pipeline interruption.

Selectivity Effect on Throughput —

We use a 6-way stream join realization and measure the maximum, average, and minimum input throughputs as we vary the match probability in Figure 6.2.5.

At match probabilities between 0.0001% and 0.001%, we observe a small slowdown in the average input throughput from $10K$ to $9K$ $tuples/s$. As the match probability further increases to 0.01% and 0.1%, the input

throughput drops to less than $1K$ and 100 $tuples/s$, respectively. Notice that Figure 6.2.5 presents the warm-up phase, while the actual throughput measurements for match probabilities of

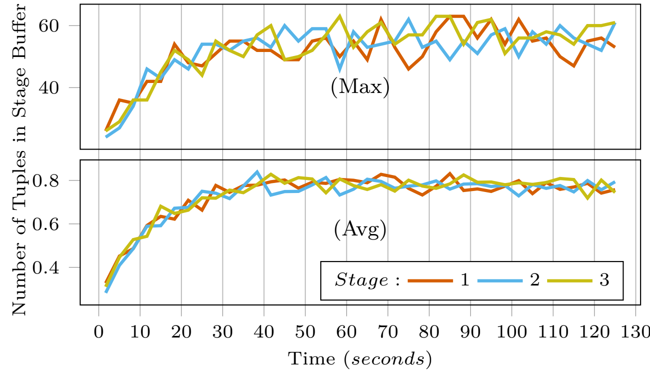


Figure 6.5.2: Stage buffer usage ($3-way, w: 2^{14}, mp: 0.1\%$).

0.0001%, 0.001%, and 0.01% settle at approximately $10K$, $7K$, and $1K$ *tuples/s*, respectively.

The drop in throughput as a function of increasing match probability is expected due to the superlinear growth in the number of intermediate results. This scenario is better demonstrated in Figure 6.5.1, where we observe a significant increase in the average and maximum numbers of tuples in the stage buffer as we increase the match probability. The reason for a large gap between the average and maximum numbers of tuples in this buffer is the burst generation of intermediate results. A join core in a pipeline stage may remain idle for a short period but can produce multiple intermediate results when it receives a new tuple (or an intermediate result) to process, while the join core in the next pipeline stage consumes the intermediate results one at a time, which forces the remaining results to wait in the stage buffer.

Steady-State Throughput Measurements —

After the warm-up phase, the behavior of Circular-MJ does not experience significant fluctuations. Figure 6.5.3 presents steady-state throughput measurements upon changing the tuples' origin distribution mean (μ) from 1 to 6 in our 6-way Circular-MJ while keeping the deviation (σ) at 1.

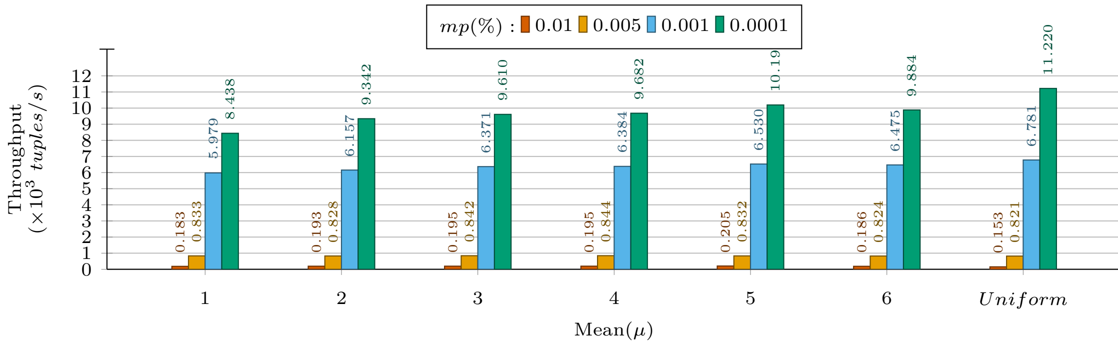
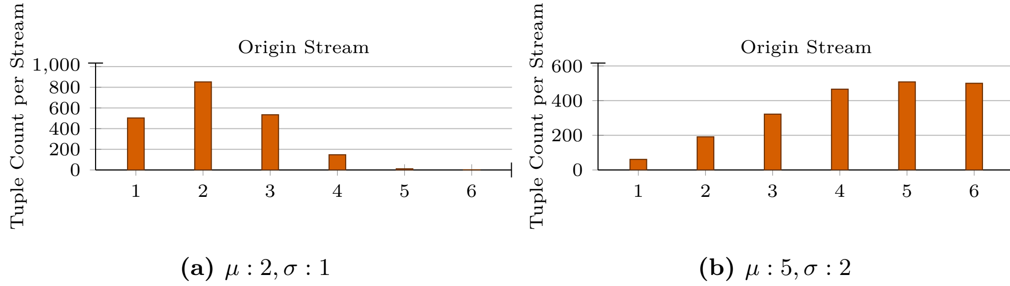


Figure 6.5.3: Steady-state throughput measurements ($6-way, w: 2^{14}, \sigma: 1$).



(a) $\mu : 2, \sigma : 1$

(b) $\mu : 5, \sigma : 2$

Figure 6.5.4: Origin arbitration using normal distribution and limiter.

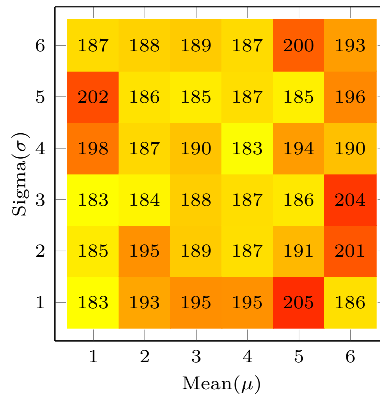


Figure 6.5.5: Steady-state throughput measurements (tuples/s) ($6 - way, w : 2^{14}, mp : 0.01\%$).

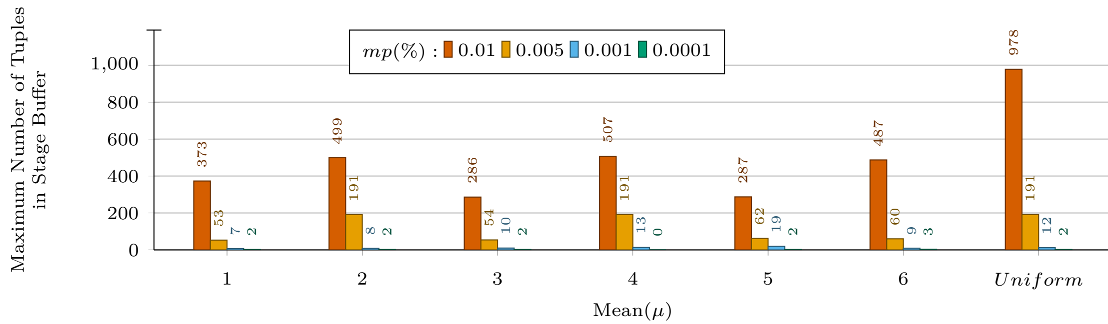


Figure 6.5.6: Stage buffer max usage ($1^{st} stage, 6 - way, w : 2^{14}, \sigma : 1$).

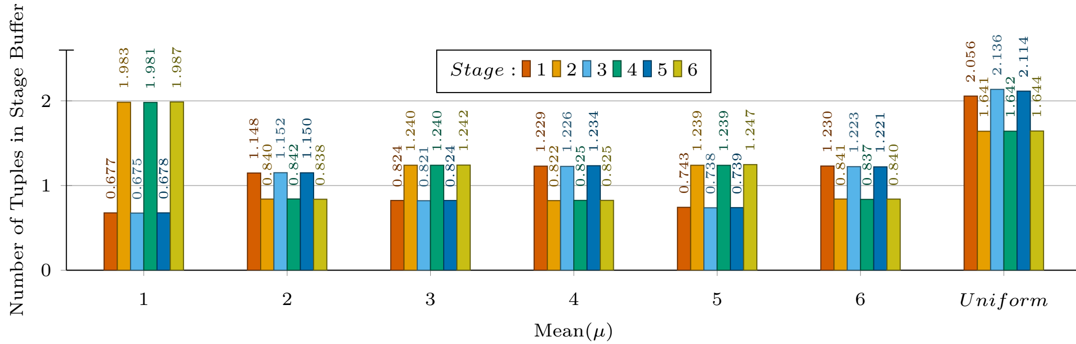


Figure 6.5.7: Stage buffer average usage for all pipeline stages ($6-way, w: 2^{14}, \sigma: 1, mp: 0.01$).

We also vary the match probabilities from 0.0001% to 0.01%. As we increase the tuples' origin distribution mean (μ) from 1 to 6, we observe a slight improvement in the input throughput when the match probability is low. This is due to the parallelism from pipelining improvements as a result of having less blockage in the early stages of the distribution chain, which allows faster consumption of new tuples assigned to the earlier pipeline stages. Note that the distribution chain spreads new tuples from the first pipeline stage to the last one, although the routing times of new tuples in the distribution chain are negligible compared to that of processing a new tuple.

The uniform distribution results in higher throughput when the match probability is low since all pipeline stages receive, on average, the same number of new tuples, which overall leads to improved parallel processing. As the match probability increases, the uniform (as opposed to the normal) distribution causes congestion in multiple pipeline stages simultaneously, which has a higher impact on the distribution of new tuples and correspondingly on the processing throughput. This scenario is also confirmed by the higher number of tuples in the stage buffer (shown in Figures 6.5.6 and 6.5.7).

To evaluate the effects of the mean (μ) and deviation (σ) on the throughput, we use a high match probability to put Circular-MJ under a heavy load (Figure 6.5.5). In the ideal case, the measurements should report the same throughput, thus demonstrating a perfect distribution of tuples to all pipeline stages. Note that in every tuple insertion, all processing units in the pipeline are involved in the execution. Therefore, in the ideal case, independent of the origin of the incoming tuples, we should see the same throughput. The measurements in Figure 6.5.5 nicely demonstrate that the added penalties by the distribution chain and neighbor-to-neighbor stage dependencies are not critical and do not heavily influence the processing throughput.

Stage Buffer Usage Analysis —

In Figure 6.5.2, we see the stage buffers' usage for all three stages of a circular 3-way stream join pipeline. Here, we use a smaller number of streams to draw a more clear comparison. We

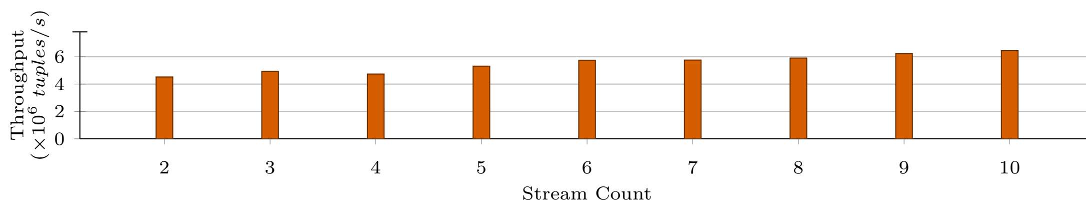


Figure 6.5.8: Steady-state throughput measurements with hash-based stream join ($w : 2^{14}, mp : 0.0001\%$).

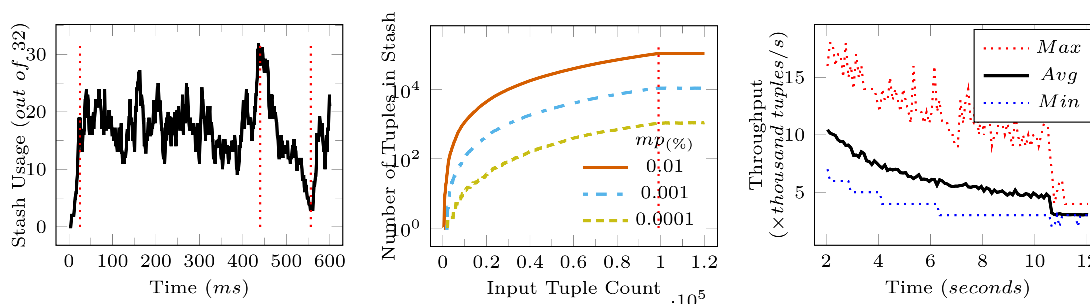


Figure 6.5.9: Stash activity ($w : 2^{10}, s : 2^5, mp : 0.001\%$).

Figure 6.5.10: Stash utilization ($w : 2^{15}$).

Figure 6.5.11: Stash overwrite effect ($w : 2^{15}, s : 2^{10}, mp : 0.0001\%$).

observe similar growth in the usage of all three stage buffers, where the average and maximum values settle at approximately 0.7 and 50 tuples, respectively. The similar usage patterns for all stage buffers shows that the circular pipeline is working in a balanced state, meaning that there are no starved or overloaded stages.

Figure 6.5.6 presents the maximum number of intermediate results in a stage buffer during the whole experiment for a 6-way Circular-MJ while changing the match probability and the normal distribution mean (μ). An increase in the match probability rapidly increases the usage due to the higher production rate of intermediate results.

In Figure 6.5.7, we report the average value for the number of tuples in the stage buffer for all six stages of a 6-way Circular-MJ. As expected, the average number of tuples remains small even for the high match probability of 0.01%. However, we observe an oscillation between odd and even stages that is common in systems with feedback as long as it remains at a small margin.

Considering both Figures 6.5.6 and 6.5.7, our measurements show that the uniform distribution of tuples' origin puts a greater strain on the circular pipeline since it spreads new tuples uniformly between all stages, which increases the chance of stalls, as also shown in Figure 6.5.3. This is evident from the higher average number of tuples that accumulate in stage buffers. Furthermore, we observe a small average value for the stage buffers' usage (Figure 6.5.7), regardless of the high burst generation of intermediate results. Therefore, Circular-MJ remains operational with smaller stage buffers than the maximum values presented in Figure 6.5.6.

Hash-Based Stream Join Measurements —

To demonstrate the applicability of our solution in the deployment of other join algorithms, we present measurements for a Circular-MJ realization that benefits from a hash-based equi-join hardware in Figure 6.5.8. Here, the input throughput mainly depends on the efficiency of the hashing algorithm. As expected, our hash-based solution presents a significantly higher throughput, which is due to the fast probing of hashed sliding windows. Similar to the nested-loop Circular-MJ, we see improvements in the processing throughput as we increase the number of streams, which is due to the increase in parallelism from pipelining.

6.5.2 Stashed-MJ Evaluations

In this section, we evaluate our optimized pipelined stream join (Stashed-MJ) with stash enabled and disabled.

Stash Activity Monitoring —

For a small stash (specified by s), the progress timeline in our Stashed-MJis demonstrated in Figure 6.5.9. It takes approximately $25ms$ for the stash to

proceed beyond its warm-up phase (specified by the first vertically dotted line). During this time, the number of intermediate results in the stash grows until it reaches a value (here 19) defined by the arriving tuples' match probability and the size of the sliding windows. At approximately $450ms$, we observe a peak in Figure 6.5.9 that leads to overwriting the stash because the number of intermediate results grows larger than its size. After the overwrite, the stash goes into the invalid state while monitoring expirations for recovery, which occurs at approximately $550ms$.

The peak in Figure 6.5.9 is later followed by a drop caused by the following sequence of events. First, the stash storage mechanism in the invalid state can overwrite valid results even when there are free slots available. Second, in our example, the full utilization of the stash was caused by a temporal increase in the match rate. Afterwards, the presence of fewer valid intermediate results in the stash and the decrease in the match rate (after the temporal increase) result in (temporally) fewer valid intermediate results in the stash and, in turn, this transient valley.

Stash Usage Analysis —

Experiments for stash usage in Stashed-MJ over a pair of $S_1 \& S_2$ streams are presented in Figure 6.5.10. For a window size of 2^{15}, the warm-up period lasts for approximately $100K$ tuple insertions, roughly equal to 3×2^{15}. The higher match probability leads to more intermediate results in the stash.

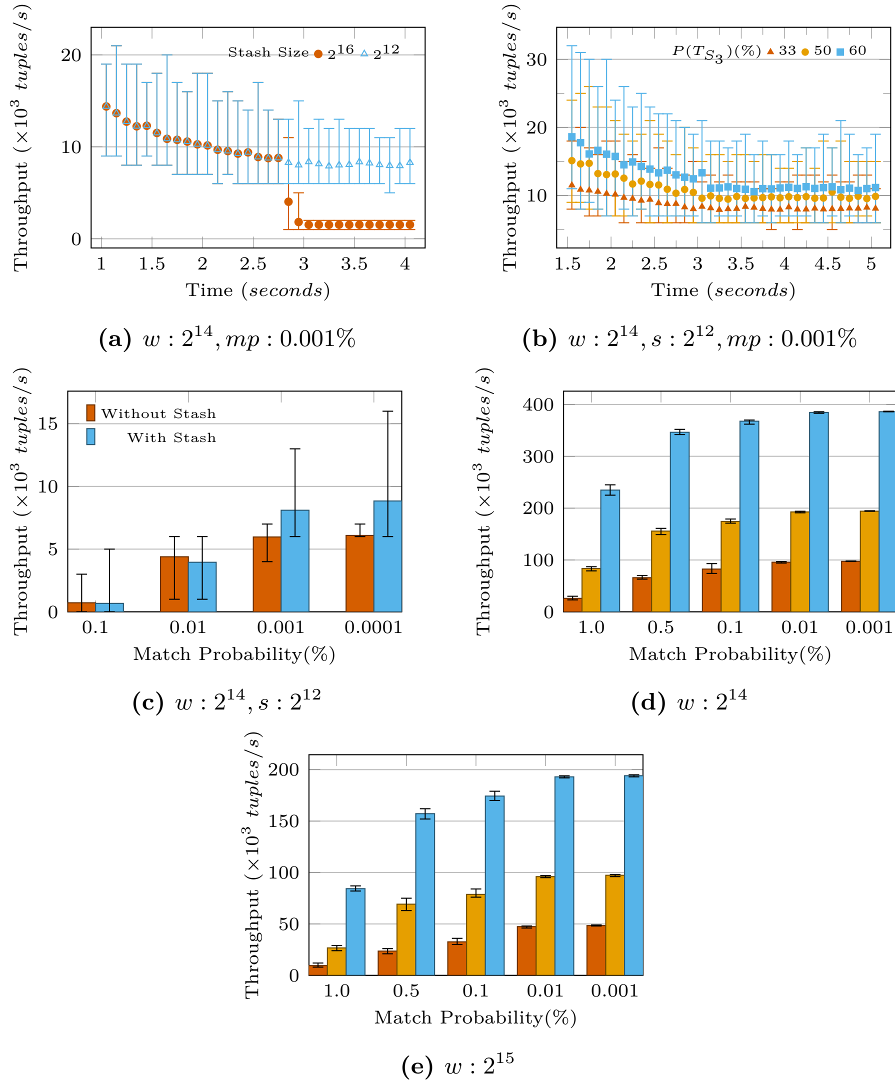


Figure 6.5.12: a) Intermittent usage issue on a large stash. b) Tuples' origin stream effect when using a stash. c) Stash (disabled vs. enabled) effect on throughput. d & e) Match probability effect in Parallel-MJ throughput ($G - JCs$: 16(■), 32(■), & 64(■)).

Stash Overwrite Effect —

The effect of an overwrite in the stash on throughput is also shown in Figure 6.5.11. At approximately $11s$ from the start of processing, while the stash is almost at the end of its warm-up period, we observe a significant drop in throughput that is due to an overwrite in the stash. Because the stash is in the invalid state, our Stashed-MJ uses the same processing path (resulting in a less variable throughput) as the other streams to process S_3, which reduces the throughput.

Excessively Large Stash Effect —

Figure 6.5.12a presents the effect of choosing a large stash on the input throughput, which leads to a significant drop after the first expiration at approximately $3s$ after the start of processing. Before the first expiration, there is no need to traverse the whole stash since there has been no intermittent usage yet. Afterwards, we need to process the whole stash, which results in a significant performance drop when the stash size is comparably larger than the sliding windows. We observe an approximately fivefold reduction of the average throughput after the first expiration when using a stash size of 2^{16} compared to 2^{12}.

Tuples' Origin Stream Effect —

In Figure 6.5.12b, we observe the effect of the percentage of reception from the S_3 stream compared to other two streams in our Stashed-MJ with stash. For example, 33% means that on average, we are receiving the same number of tuples from all three streams. Each sample in this diagram presents the maximum, average, and minimum values of 100 samples. Since there are no stashes for the other stream pairs, tuples from the S_1 and S_2 streams have to be processed against other sliding windows, while tuples from the S_3 stream are only evaluated against the intermediate results available in the stash. This is the reason for the large gap between the maximum and minimum throughputs, as S_3 tuples are processed much faster than the others.

Steady-State Measurements —

The steady-state throughput measurements for our Stashed-MJ with and without stash are presented in Figure 6.5.12c. For match probabilities higher than 0.01%, we do not see any improvement from the stash; however, as we reduce this probability, the stash becomes more effective. A match probability of 0.01% and a window size of 2^{14} produces more results than a 2^{14} (window size). Therefore, the stash stays in the invalid state permanently, meaning it will be ignored by the pipeline. We see a slight degradation in throughput from the pipeline without stash, which is due to the stash circuitry overhead. For smaller match probabilities for the same window size, we observe a significant boost in throughput that is directly proportional to the inverse

One interesting observation is the slightly lower input throughput of Stashed-MJ (especially when

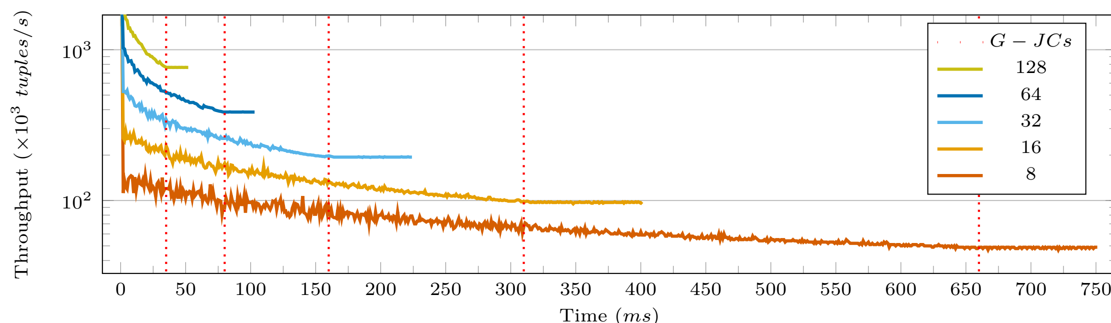


Figure 6.5.13: G-JC count effect on throughput ($w: 2^{14}, mp: 0.001\%$).

disabling the stash) compared to a 3-way Circular-MJ as demonstrated in Figures 6.2.4 and 6.5.12c, respectively. The reason for the lower throughput is that Circular-MJ utilizes three processing units compared to two for Stashed-MJ. Therefore, Circular-MJ benefits more from parallelism due to pipelining, resulting in slightly higher input throughput.

6.5.3 Parallel-MJ Evaluations

Next, we present experimental evaluations of our parallel stream join built by integrating Stashed-MJ (without stash) into the uni-flow parallel stream join architecture, referred to as Parallel-MJ. The integration of Circular-MJ into the uni-flow parallel architecture follows analogously but is omitted in the interest of space.

Global Join Core Count Effect —

Figure 6.5.13 illustrates the throughput measurements as the number of G-JCs (G-JCs) increases. In each diagram, we see a warm-up period in which the system throughput drops as the sliding windows fill up. The end of the warm-up period is shown by the dotted vertical line in the figure. The measurements show a linear acceleration. Using 8 G-JCs, we obtain a throughput of $\sim 47K$ $tuples/s$, which is approximately $8\times$ higher than $\sim 5.97K$ $tuples/s$ as observed in Figure 6.5.12c. Similarly, utilizing 16, 32, 64, and 128 G-JCs leads to an average throughput of $97K$, $194K$, $386K$, and $763K$ $tuples/s$, respectively.

Throughput Sensitivity to Match-Rate —

The spikes in the diagram (see Figure 6.5.13) highlight the sensitivity of the input throughput to the match-rate (mr), which is especially noticeable in our solution due to the propagation of intermediate result bursts to multiple pipeline stages. For example, having two matches (intermediate results) in the first pipeline stage drops the throughput to half compared to having one match.

We observe that by increasing the number of G-JCs, the number of spikes decreases significantly for two reasons. First, the division of windows into more sub-windows decreases the number of matches (intermediate results in the first stage) in each G-JC. The second reason is the overlapping of processing time for those matches in independent G-JCs. For example, assume we expect four matches on average in the first pipeline stage. Using two G-JCs, we will have two matches on average in each G-JC, and we only observe the time for processing two matches since the two G-JCs operate in parallel.

Match Probability Effect —

Figures 6.5.12d and 6.5.12e present throughput measurements for Parallel-MJ realizations with 16, 32, and 64 G-JCs as we increase the match probability from right to left. For stream joins on more than two streams, the increase in the match probability increases the number of resulting tuples, which leads to saturation of the result collection circuitry. This limits the input throughput by means of feedback. However, this is not the case in our pipeline solution since the number of matches in the first stage is multiplied by the processing time in the second stage, which limits the throughput. This means that each generated intermediate result in the first stage has to wait for the processing of the preceding results in the second stage.

Chapter 7

Simplex Stream Processor on a Custom Network-on-Chip

Stream processing acceleration is driven by the continuously increasing volume and velocity of data generated on the Web as well as the limitations of storage, computation, and power consumption. Hardware solutions provide the best in terms of class performance and power consumption, but the relevant solutions are hindered by the high cost of research and development, and the long time needed to market the applications. In this work, we propose our simplex stream processor (SSP), a complete rethinking of a previously proposed customized and flexible query processor that targets real-time stream processing. SSP uses unidirectional dataflow not dedicated to any specific type of stream. Therefore, its processing components are simply placed one after another on a single general data path that facilitates query (operator) mapping.

In SSP, the concepts of the distribution network and processing components are implemented as two separate entities connected to each other using generic interfaces. This approach allows for the straightforward adoption of a proper architecture for a family of queries, rather than forcing a relatively rigid chain of processing blocks to implement the queries.

Our experimental evaluations of the third query mapping of the TPC-H without further optimization yielded processing times of 300, 1220, and 3520 milliseconds for data streams with scale factor sizes of one, four, and 10 GB, respectively.

This section presents the following contributions:

1. We propose a stream customized network-on-chip (SCNoC) with instruction sets for routing

instructions and tuples.

2. We propose SSP with a configurable topology (placement of processing components in the distribution network) and instruction sets built on our SCNoC.

3. We propose a hash-based stream join (HB-SJ) architecture for fast equi-join processing and integrate it into a modified version of our circular multiway stream join (Circular-MJ) architecture.

4. We present a hardware architecture for the aggregation and groupby operators.

5. We present a mapping of and implement the TPC-H on a new instance of SSP and evaluate it based on certain performance metrics.

7.1 SSP Architecture

We first present our vision for SSP and then describe the complete system stack, including software and hardware stacks for data handling and processing.

7.1.1 System Vision

The complete software/hardware components of SSP are shown in Figure 7.2.5. When working with a specialized hardware, filtering unimportant (for current query processing) parts of data is crucial to delivering a practical system. However, these parts need to be reattached to the resulting tuples. We assume that we have an age attribute in our tuples that is not used in the given query; however, we need this attribute in the resulting tuples. To avoid using valuable hardware resources to transfer this attribute, we need to cut it from the input tuples and reattach it to the corresponding resulting tuples.

The components to perform this task are shown in the upper part of Figure 7.2.5. We decompose each tuple into necessary and unnecessary segments using a *decomposer*. The former are passed to the SSP hardware for processing while the latter are stored in a table-like data structure. After the processing, the resulting tuples receive their detached segments using the *composer* component, and the complete resulting tuples are sent to the system output.

SSP is designed to be flexible enough to accept new queries or updates to existing queries as long as they follow the logic of the chosen SSP topology. The *query assigner* component, which is responsible for inserting query plans, and the SSP hardware are shown in the middle of Figure 7.2.5. After receiving a new query, the *query assigner* maps it to the available processing blocks and

generates the corresponding instructions to program the query. These instructions are fed into the SSP hardware through the input port to the *data streams*.

As the last part of our system, the *SSP designer* provides the flexibility to design a custom topology in a graphical user interface by using the provided component libraries without digging into the details of hardware design and development. Following topology design, it is first synthesized and then programmed into the FPGA.

7.1.2 Topology Brick

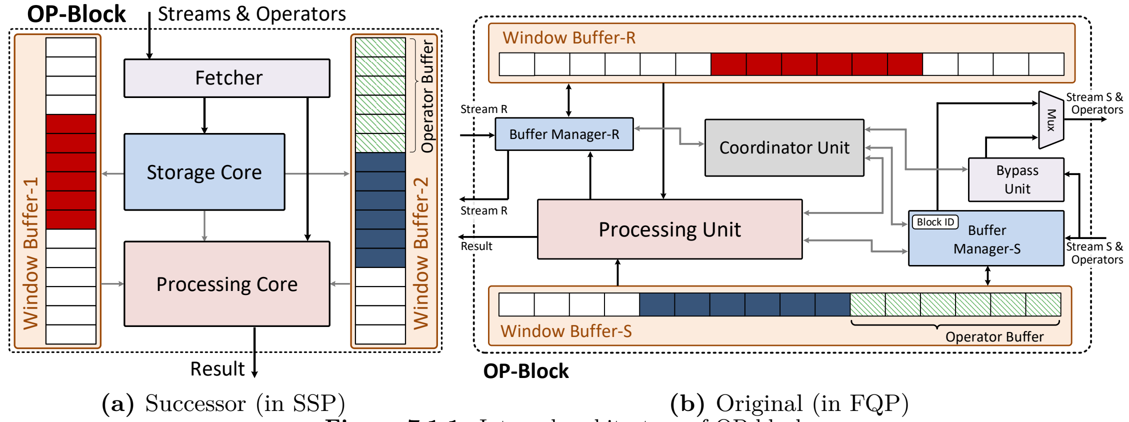
The processing components of SSP are packed, in addition to the local data distribution and collection components, into specific sets that we refer to as *topology bricks*, shown in the lower part of Figure 7.2.6. The way in which the topology bricks are connected defines the topology of an SSP instance that characterizes the capabilities and properties of the system. Each topology brick contains two SCNoC components of LSwitch and Collector, in addition to a set of data handling and processing components specified by the *bypass* and *PUnit 1..N* labels in Figure 7.2.6.

7.1.3 Data Handling and Processing Blocks

The processing units/components (PUnits) follow a straightforward protocol for receiving data/instructions from a single-input port and push the resulting data from their single-output port. They can implement any type of execution engine, ranging from simple filtering units to highly specialized hardware or even general-purpose processors (i.e. ARM cores). PUnits can have their own dedicated memory or use a shared memory hierarchy with other PUnits through a dedicated port based on the application requirements.

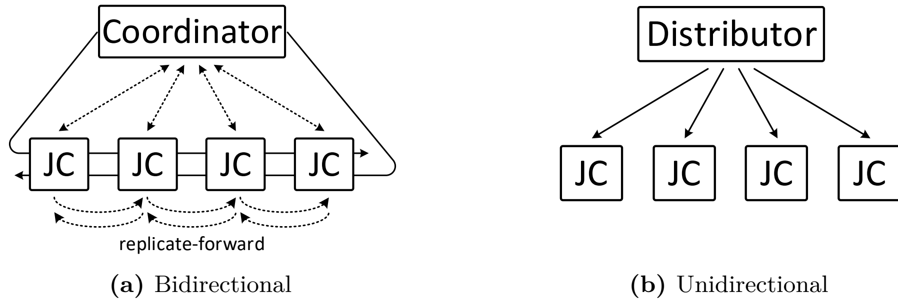
7.1.4 OP Block

Although the SSP architecture supports and even motivates the use of general-purpose processing cores (i.e. ARM cores) to handle small but control-intensive tasks within queries, we also have the option of using custom-programmable processors for the dataflow processing pattern of the streams. As an example of such processors, we present our OP-Block that is redesigned based on the unidirectional dataflow processing model, and is shown in Figure 7.1.1a. The main challenge in the design of OP-Block is the processing (and parallelization) of the resource-intensive join operation in stream processing. The current realization of OP-Block benefits from a dedicated memory storage to realize the sliding-window buffers needed for the join operation, which are also used as instruction storage.



(a) Successor (in SSP) **(b)** Original (in FQP)

Figure 7.1.1: Internal architecture of OP block.



(a) Bidirectional **(b)** Unidirectional

Figure 7.1.2: Stream join dataflow models.

The original OP-Block and the redesigned (successor) OP-Block are illustrated in Figures 7.1.1b and 7.1.1a, respectively. In the redesigned model, data pass in a single top-down flow, in which an OP-Block receives tuples directly from the LSwitch. Each OP-Block is placed inside a PUnit and operates independently of the other PUnits. The redesigned architecture fully utilizes the communication bandwidth provided because all tuples travel through the same path to the *processing core*, unlike the double-path architecture of the original OP-Block. Therefore, regardless of the incoming tuple rate for each stream, every tuple has access to the full bandwidth. The same concept applies when using more OP-Blocks (PUnits) to parallelize a processing task, as shown in Figure 7.1.2b.

In the design of FQP the OP-Blocks are connected to one another as in the model shown in Figure 7.1.2a, which, in addition to complicating operations on more than two streams, significantly degrades performance owing to the underutilized communication bandwidth. To solve this problem for the original OP-Block, assume that we are receiving tuples only from stream R; then, all communication channels for stream S are left unutilized. Even with an equal tuple rate for both streams, it is impossible to achieve the simultaneous transmission of both T_R and T_S between neighboring join cores due to the locks needed to avoid race conditions.

Comparing the internal design of an OP-Block, utilized as a join core based on the original model, Figure 7.1.1b, with one based on the redesigned architecture, Figure 7.1.1a, we observe

a significant reduction in the number of internal components that correspondingly reduces the design complexity. Neighbor-to-neighbor tuple transfer circuitries for two streams are eliminated from the *buffer manager-R & -S* and the *coordination unit* components. They are reduced and merged to form the *fetcher* and *storage core* components in the redesigned architecture. This improvement reduces the number of input/output ports from five to two, which significantly reduces the hardware complexity because the number of input/output ports is often an important indication of the complexity and final cost of hardware design. We omit further description of the supported operations and their instruction sets as they are similar to the ones presented in [67].

7.1.5 Complementary Custom Blocks

We briefly describe the important processing and data handling components that we designed and implemented (in VHDL), and then customized for our case study benchmark on our SSP, which is the TPC-H third query presented in Section 7.3.1.

Multiway Stream Join —

Considering the crucial role of joins as among the most resource-intensive operators in relational databases, it is not surprising that stream joins have been the focus of considerable research on data streams [53, 39, 85, 43, 28, 100, 54, 69, 44]. For example, consider TPC-H [30], where 20 queries (of 22) contain a join operator, and 12 of them use multiway joins, some up to seven joins. However, the importance of joins is no longer limited to only the classic relational setting. The emergence of the Internet of Things (IoT) has introduced a wave of applications that rely on sensing, gathering, and processing data from an increasingly large number of connected devices. These applications range from the scientific and engineering domains to complex pattern matching methodologies [14, 58, 73].

In SSP, we utilize an instance of our Circular-MJ, presented in [65], placed inside a PUnit to support multiway stream joins when needed. Circular-MJ benefits from a scalable pipelining mechanism for processing. The abstract architecture of Circular-MJ is shown in Figure 7.1.3.

In the design of Circular-MJ each join core (operator) is connected to only one sliding window with two entries. Each core receives its new tuples determined based on their origin from its entry that is connected to a *distribution chain.* Each join core's side input ports are placed in a circular data path that carries intermediate results from one join core to the next. The resulting tuples are emitted after processing a new tuple in exactly $N-1$ cores, where we have N streams. The remaining core is responsible for storing the new tuple in its sliding window.

The Circular-MJ pipeline has the same number of stages (join cores) as input streams. Each stage is placed between isolating sets of registers, and is responsible for processing a new tuple against a specific sliding window. When the window in a stage belongs to the origin of the given tuple, the

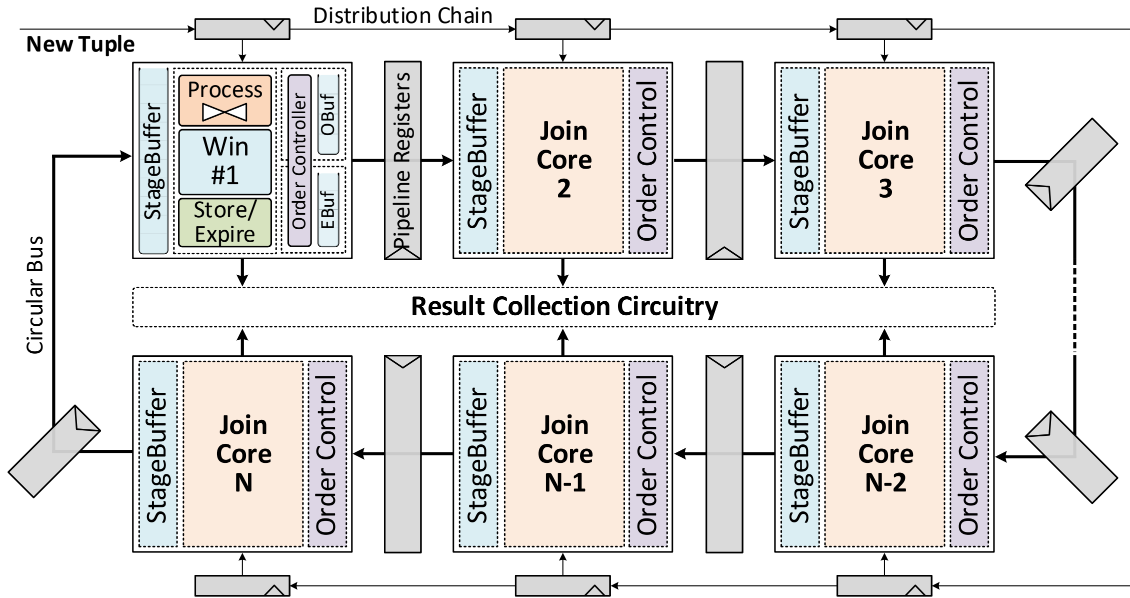


Figure 7.1.3: Scalable multiway stream joins (Circular-MJ).

store and expiration tasks are performed instead of processing.

In a general Circular-MJ, there is a relation between each of two streams that requires that the circular design has a scalable architecture; however, in the TPC-H third query, the *lineitem* and *customer* streams are both related to different attributes of the *orders'* stream. This property of this query allows for further customization of our Circular-MJ.

Hash-based Stream Join —

Using a nested loop algorithm, although sufficient for the proof of concept of our SSP makes it impossible to execute the TPC-H benchmarks in a reasonably short time because of their sizes. To accelerate execution, we propose a hardware hash-based stream join solution that, as the name suggests, benefits from a customized hashing mechanism to process equi-joins.

Figure 7.1.4 presents the abstract architecture of our hash-based stream join unit. It shows only the unit used to process a stream (e.g. R) against an opposing stream (e.g. S). For a complete join, we need to duplicate this architecture for processing in the other direction. Alternatively, we can use the unit in Figure 7.1.4 in each of the join cores in the Circular-MJwhere, depending on the number of stages, we can perform anywhere from two-way up to N-way stream joins.

Our hash-based stream join unit, shown in Figure 7.1.4, utilizes two Murmur3 hash functions with four storage tables. If the selected index (row) is full in all four tables, the new tuple is inserted into an *overflow buffer*. To preserve the order of arrival of the tuples (necessary in the

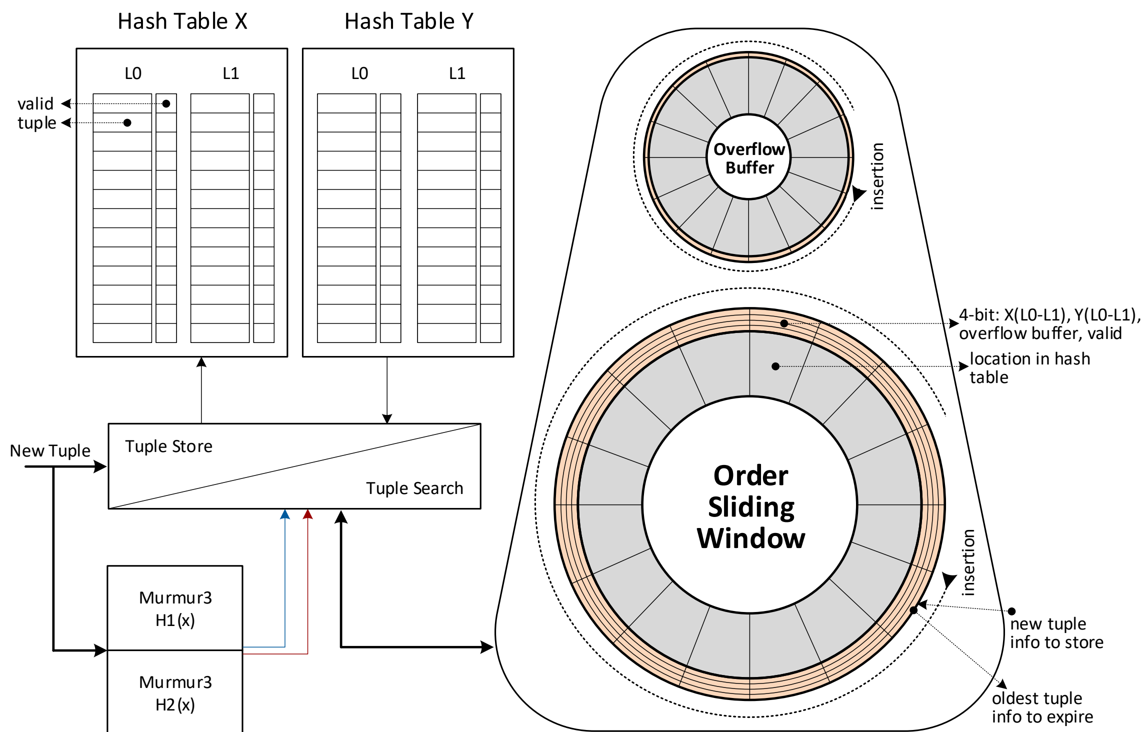


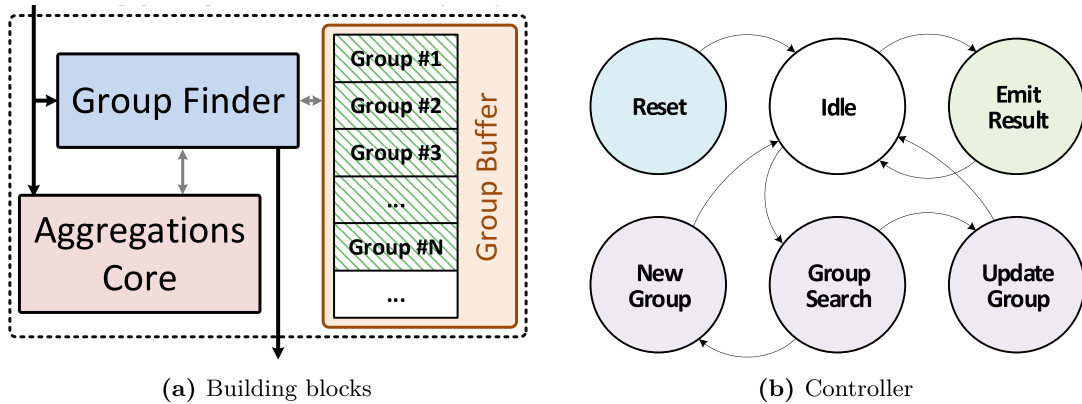
Figure 7.1.4: Hash-based architecture for stream join (HB-SJ).

count-based sliding window), our proposed solution uses an *order sliding window*. The order sliding window preserves the location of each new tuple storage (one of hash tables or the overflow buffer) in addition to the hash table index. The expiration task occurs before the insertion of a new tuple by removing the oldest tuple, determined by the oldest tuple in the order sliding window. For a new tuple search, the hash table key is calculated first and then all four hash tables are examined in parallel against the join condition to find their matches. At the end of the operation, the overflow buffer is searched using a nested loop join to find matches that are not stored in the hash tables. Although the overflow buffer is relatively small, we can use a faster algorithm for the search to accelerate processing as it is still the most time-consuming operation in our HB-SJ.

The architecture of the time-based sliding window does not contain the *order sliding window*, and expiration is performed on a new tuple storage and a new tuple search. Moreover, further adjustments in the number of hash functions and tables, and the use of other extensions (i.e. Cuckoo hashing scheme), are beneficial for the efficiency of hash tables, although they remain beyond the scope of SSP.

Aggregation-GroupBy —

In queries, it is common to have a grouping operator that is based on some specified fields while



(a) Building blocks

(b) Controller

Figure 7.1.5: Aggregation-GroupBy unit.

the other fields are aggregated using one or multiple arithmetic operators. Because the aggregation is performed on some elements of each group, it is more efficient in the hardware design to combine these two (aggregation and groupBy) operators into one unit, as shown in Figure 7.1.5a.

Figure 7.1.5a shows the building blocks of the aggregation-groupBy unit and Figure 7.1.5b presents a simplified state diagram of the controller. On the insertion of a new tuple (or the intermediate result), the *group finder* unit searches the *group* buffer to find its relevant group. When a matching group is found, the aggregation task is performed (in the *aggregation core*) on that group and the new tuple. The resulting data are inserted into the group buffer. If no match is found, a new group is created and filled with the new tuple.

The architecture of the order by operator is similar to that of the aggregation-groupBy unit but with no aggregation core. In the order by operator, the group finder searches for the correct location of each new group and inserts it. We use bubble sort to insert each new group into its corresponding sorted location.

7.1.6 Synchronizer Blocks

When merging multiple data paths for the processing of a query or set of queries, a synchronizer component is needed, and is placed in one of the PUnits. A common approach to synchronization is to use tokens generated in a component, where the correct order of tuples (e.g. from multiple streams) relative to one another is available. Later in the processing path, these tokens are used to restore the correct order of tuples when multiple data paths meet one another.

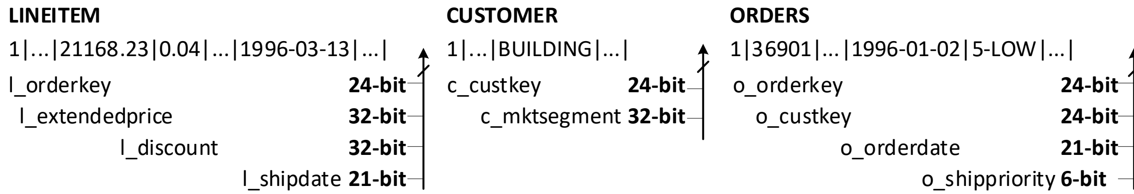


Figure 7.1.6: Tuple-to-bit conversion.

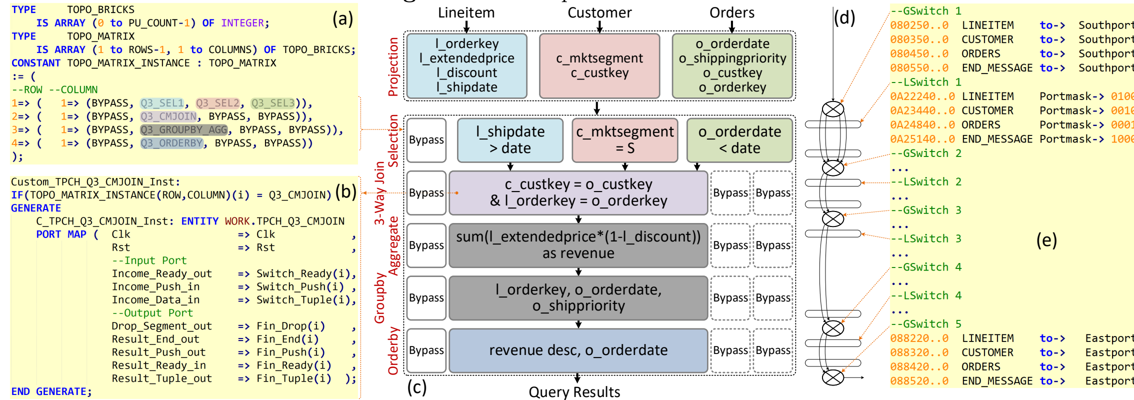


Figure 7.1.7: TPC-H third query mapping on SSP.

7.1.7 Parallel Processing

Because the SCNoC distributes incoming tuples without changing the order of arrival, we can use parallelization inside or across multiple topology bricks. The processing throughput for such stateless operators (i.e. selection) is already high on hardware. However, we can still parallelize them using a block to reindex a stream into multiple sub-streams that can be divided by the next LSwitch for distribution among separate processing blocks.

For more complex state operators such as join, in our parallelization technique, each processing block performs the requested task based on its location in the SSP topology. As an example, in stream join parallelization, the architecture of SplitJoin [68] can be implemented both inside and across topology bricks. For parallelization inside topology bricks, a LSwitch replicates and distributes the incoming streams to all processing units responsible for the join operation, similar to what is shown in Figure 7.1.2b. Each PUnit behaves as a separate join core and, depending on its position among the join cores, stores one of multiple received tuples in its respective sub-sliding window, while all join cores perform the search task on each incoming tuple as presented in [68].

7.2 SCNoC Architecture

A processing engine that integrates all or most of the necessary components to perform a task is referred to as a system-on-a-chip (SoC). SoCs have the best in-class performance and

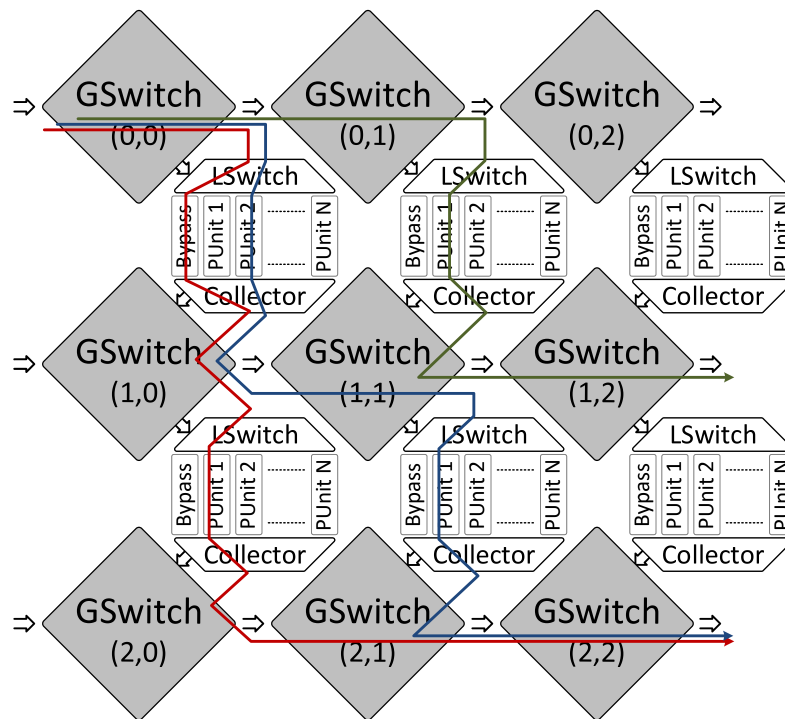


Figure 7.2.1: SCNoC architecture.

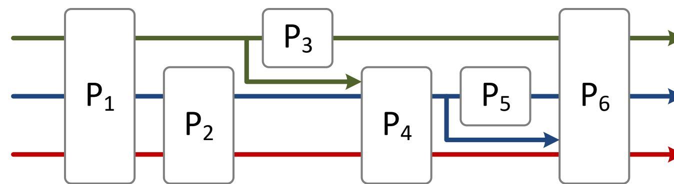


Figure 7.2.2: SCNoC flow model.

efficiency because they bring these components close to one another, removing most unnecessary intercomponent interfaces and their added latency. SoCs are particularly interesting in data stream handling owing to the high velocity and volume of streams because every added latency to the processing path superlinearly increases the cost of the system.

7.2.1 Background

Each SoC solution benefits from a particular type of communication sub-system to transfer data between various components, such as a processor, a coprocessor, a DMA[1] controller, and peripherals.

[1]Direct memory access.

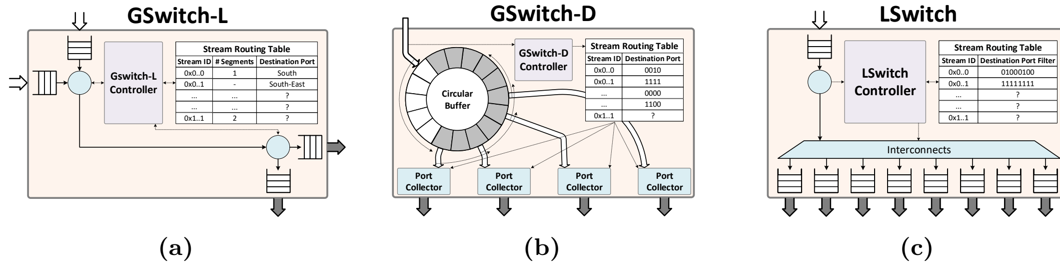


Figure 7.2.3: SCNoC communication blocks.

SoCs commonly use a *bus* (e.g., the ARM AMBA[2]) system that acts as communication sub-system. A bus consists of an access controller and a set of wirings that connect all internal components of an SoC.

Despite the simplicity of the conventional bus architecture, it fails to scale when the number of connected components increases. Thus, we are forced to rely on distributed architectures, e.g. network-on-chip (NoC). NoC solutions have their own challenges and opportunities. They provide a substantial bandwidth advantage over bus sub-systems, but at the expense of extra controlling logics (terms of both hardware and software) to ensure consistency of communication.

7.2.2 Main Drawback

Typical NoCs are designed for general-use cases to handle a wide range of workloads. However, this generality is not needed, or can at least be reduced in stream processing in favor of simpler and more efficient communication circuitries, resulting in better efficiency and performance characteristics.

A data stream is a (potentially unbounded) sequence of tuples that flow one after another. This flow-based property of data streams motivates a processing architecture that avoids circulating data between the internal components. As an example, filtering the resulting data from an aggregation unit using a selection unit, and then sending the filtered data back to the same aggregation unit (to perform another type of operation) leads to circulation that causes major issues in real-time processing. A circular path in a digital system, constantly receiving new data, inherently adds the possibility of deadlock that necessitates complex buffering and communication circuitries to reduce the chance of congestion.

To benefit from the flow-based property of streams, our SSP utilizes a stream -customized network-on-chip (SCNoC), as illustrated in Figure 7.2.1.

[2] Advanced microcontroller bus architecture.

7.2.3 Design Rationale

A major decision that heavily influences our SCNoC architecture is to limit internal (interblock) communication to unidirectional rather than bidirectional communication. Although this eliminates the possibility of returning data to a former processing block, it leads to a data path suitable for real-time processing by forbidding inefficient query mappings. In other words, the design of our SCNoC targets a paradigm where the processing is in the flow of streams. It is possible to add feedback paths to the SCNoC architecture by using a branch component (Figure 7.2.1) instead of a processing unit (PUnit) and feeding the output port to one of the open entrances of the SCNoC architecture. However, this approach is not recommended due to additional complexities, such as race conditions. If such a feedback is necessary in the processing of a query, the approach advised for SSP is to implement the entire feedback part (including its feedback path) in a single PUnit.

Figure 7.2.1 presents a 2D implementation of our SCNoC including some of its main components such as GSwitch-L and LSwitchthat perform the main data routing tasks. After implementing a topology of SCNoC that is reconfigurable based on the application requirements, the routing instructions are fed to SCNoC to program GSwitch-Ls and LSwitchs. Subsequently, streams of data are routed and brought to their corresponding PUnits by the GSwitch-L and LSwitch components.

The processing flow model supported by our SCNoC (Figure 7.2.1) is illustrated in a linear model in Figure 7.2.2. In this model, we examine a set of flowing streams, and place processing units and branches on them to produce the final results of a query.

7.2.4 GSwitch

SCNoC is constructed from programmable (global) switches that we refer to as GSwitch. We propose two architectures for a GSwitch, specified by -L and -D. We provide in-depth descriptions of its design and implementation for GSwitch-L because it is used throughout this work and in the benchmarks. However, we only briefly introduce the properties of GSwitch-D and leave further descriptions of its design and explorations of its performance to future work.

GSwitches are responsible for bringing data to a specific set of PUnits. The main properties of the design of a GSwitch are 1) data pipelining to mitigate processing stalls, and 2) the number of flows to be routed, or more precisely, the number of input ports (x) routed to a number of output ports (y).

GSwitch-L —

GSwitch-L is a variant of GSwitch with two input (x=2) and two output (y=2) ports, as shown

in Figure 7.2.3a. This unit has a *stream routing table* that indicates the output port of each stream. The routing information in this table is programmed using its corresponding instructions. Each row in this table belongs to a stream and contains three fields: 1) *stream ID*; 2) *# segments*, which specifies the size of each tuple in terms of number of chunks;[3] and 3) a *destination port*, which can be one or both of the output ports. The *GSwitch controller* monitors incoming tuples from both input ports, and, when there are instructions related to this unit, it updates the table based on the tuples, and then pushes them to the output ports to reach their target blocks.

The input and output buffers reduce stalls in the flow of data because they allow for the insertion of new data even when the processing blocks in the outputs are busy. Therefore, these buffers improve pipelining parallelism. In the other words, the input and output buffers allow the *GSwitch controller* to route currently available data ahead of time without waiting for the processing blocks in the output.

GSwitch-D —

We refer to a GSwitch design customized for distribution as GSwitch-D, which is preferable when our application requires large data broadcasts, i.e. at the input of SCNoC where streams are distributed according to their selected processing paths. The abstract architecture of the GSwitch-D (x=1, y=bounded n) is shown in Figure 7.2.3b. The design consists of a *stream routing table* that preserves stream IDs with their corresponding masks. There is a one-to-one assignment between each bit in the mask field and its corresponding output port, and each port's bit determines whether a stream must be sent out to that port. The masking information is programmed through an SCNoC instruction used by the *GSwitch-D controller*.

In hardware, elements of the internal memory are limited and expensive resources. An interesting design aspect of GSwitch-D is the use of a shared circular buffer for all ports to reduce memory usage. Using a first-in, first-out, (FIFO) policy, each *port collector* benefits from the larger buffer, which adds more elasticity to the processing data path. Using this shared buffer, we reduce the chance of resource starvation in some GSwitch-D output ports while overloading the processing units in the other output ports, which improves processing efficiency and performance.

Compared with GSwitch-L, which has two input and two output ports, GSwitch-D is more suited to large fan-outs, where a single input is fed to many other units.

7.2.5 LSwitch

Each GSwitch-L is connected to the processing and data handling blocks through a local switch referred to as LSwitch. Figure 7.2.3c illustrates the internal building blocks of LSwitch. This unit

[3] The size of each chunk is defined statically (e.g. 64 bits) in our hardware specifications.

utilizes a programmable $1 - to - N$ switch to distribute tuples and instructions to blocks connected to its output ports. The *stream routing table* in LSwitch has two fields: 1) *stream ID* and a 2) *destination port filter*. LSwitch uses a masking technique to specify the port(s) of egress for each stream. For this reason, the destination port filter uses one-bit mask per port, which enables data transmission over that port in the same manner as in GSwitch-D. The *LSwitch controller* fills this table as it receives instructions from the LSwitch input port. If an instruction does not belong to the given LSwitch, this controller broadcasts it to all of its output ports.

LSwitch is designed to support up to N blocks in its output, where the value of N is determined based on the application requirements. *Interconnect* in Figure 7.2.3c is the main component affected by the size of N. Using a small interconnect can lead to the underutilization of SCNoC because the number of processing components (limited by N) is not sufficient to fully utilize the bandwidth provided by components of the communication network. Further, aiming for a large interconnect can negatively affect the frequency of the working clock[4] of the SCNoC, especially when parallel processing on a large number of processing blocks is needed. One can exploit multiple GSwitch-L units to instantiate multiple LSwitch units, each with an appropriate interconnect size.

7.2.6 Collector

Following the distribution of the input tuples to PUnits and other data handling blocks by LSwitch, an Collector (Figure 7.2.6) gathers the resulting tuples from them. Collector can use a hierarchical or linear architecture to gather the data, each with its advantages and disadvantages. The former architecture is suitable for a relatively large number of processing blocks as the collection task can be performed in parallel for all PUnits. The latter architecture favors topology bricks with fewer blocks because it gathers the resulting tuples one after another from them, which leads to poor scalability in relation to the number of blocks; however, it requires fewer resources to operate.

If a processing block has resulting tuples that are wider than the input, they are broken into smaller (as wide as the input) segments. Collector needs to ensure that it gathers all segments of a tuple from a block before starting the collection from another block. This is done by assigning an unused (NULL) stream ID to all segments of a tuple except the first one, which carries its origin stream ID. Whenever Collector starts the collection task from a block, it continues to gather all segments of a tuple one after another until it observes a segment with an existing stream ID. It then starts collecting a tuple from another block based on the defined priorities.

[4]A major parameter of the hardware that has a direct relation to processing performance.

7.2.7 Instruction Set

As mentioned in the foregoing, the challenge in the design of SCNoC is to maintain its architecture as simple and lightweight as possible while providing sufficient routing flexibility to bring streams to their corresponding processing blocks. In this regard, the GSwitch-L and LSwitch units are programmed using unique instructions of their own, as shown in Figure 7.2.4.

At the beginning of each tuple, there are a few bits, e.g. two or more bits, depending on the maximum number of supported streams, that specify the ID of each stream and the type of data (instruction or tuple). For example, a value of 1 in the stream ID field (Figure 7.2.4) defines the given data as a network instruction, whereas 0 defines these data as a processing block instruction. Other values are used to specify data stream IDs.

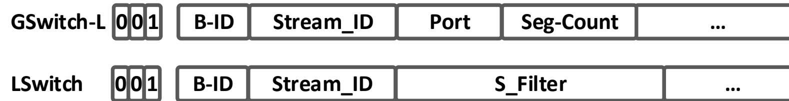


Figure 7.2.4: GSwitch-L and LSwitch instruction sets.

The instruction for GSwitch-L consists of the following: 1) *B-ID* that defines the ID of the target block for this instruction, 2) *stream-ID* that specifies the stream in which this instruction carries its routing information, 3) *port* that determines the mask of the egress port for the specified stream, and 4) *seg-count* that indicates the number of segments for each tuple for the specified stream. The remaining fields are undefined, and can be utilized in future extensions. As an example, data with fields "1 : 23 : 4 : *south* : 3 : ..." programs a GSwitch-L with a block ID of 23 to route 3 Segment-size tuples of a stream with ID 4 to its *south* port.

Similarly, the LSwitch instruction contains the *B-ID* and *stream-ID* fields, but it uses *S-filter* to specify the egress port(s).

7.2.8 Network Interface

An inherent problem in custom hardware is its sensitivity to data size. For example, when we design and build hardware for 64-bit-wide data, this hardware cannot process wide data without considering them in the initial design. The communication network in SCNoC is designed to accept wider data units by breaking the data into smaller segments. However, resource over-provisioning results in a significant increase in cost as the width of the data increases, which renders it impractical.

For a large practical solution, we propose using parallel-to-serial and serial-to-parallel network interfaces (NIs) to avoid the substantial amount of wiring otherwise needed between internal blocks. Figure 7.2.6 illustrates these interfaces in triangle pairs, where an NI converts the parallel data into

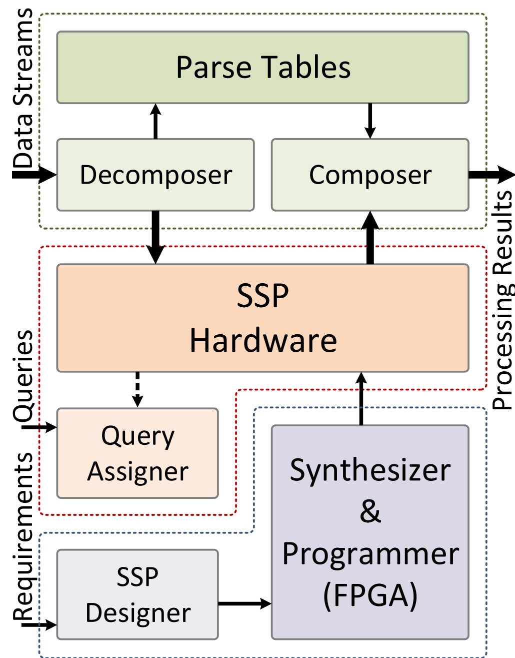


Figure 7.2.5: SSP complete system.

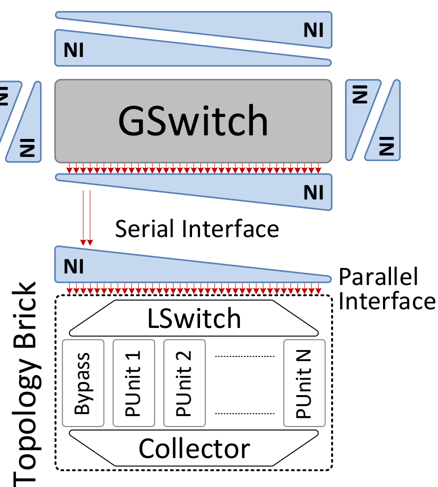


Figure 7.2.6: Communication interface.

serial while the other performs the reverse operation. The optimum number of serial lines between each pair of NIs can be determined based on the application and the technological specifications used. For example, when the processing effort and time in a PUnit are significant, the results come one after another in large intervals, which allows the use of fewer serial lines. Having more transfers necessitates the use of more serial lines. When the full bandwidth is required, it is advisable to use parallel communication rather than serial communication.

7.3 Experimental Results

We prototyped `SSP`, including its `SCNoC` and all required custom blocks, in VHDL. To extract the hardware properties, we synthesized and implemented our solution on a VCU110 development board featuring the Virtex UltraScale XCVU190-2FLGC2104E FPGA. We used Xilinx `Vivado Design Suite (ver. 2017.2)` for the synthesis and analysis of the HDL design. In the implementation, we used module analysis[5] [3], also referred to as out-of-context implementation, allows us to analyze a module independent of the remainder of the design to determine the resource utilization and extract the timing analysis. For functional evaluations, we used the `Questa Advanced Simulator` to perform cycle[6]-accurate[7] measurements.

We used the TPC-H DBGen tool [30] to generate the benchmarking tables, and used our parser (written in Python) to decompose and parse rows of tables into tuples of streams to feed into `SSP`. DBGen allows us to roughly choose the size of the database (tables) by a scale factor (SF) parameter. In this work, we used SF=1 (equivalent to a database size of 1 GB) as default, unless otherwise stated. In the following, we present a prototype of the TPC-H third query on the `SSP` framework for the evaluations in this work.

7.3.1 TPC-H Third Query Prototype

To evaluate the functionality and properties of our `SSP`, we describe in detail the query mapping (to the processing blocks) and programming steps to implement the TPC-H third (shipping priority) query. This query retrieved the shipping priority and potential revenue, defined as the *sum of*

[5]The module analysis flow implements a module out-of-context (OOC) of the top-level design. The module is implemented in a specific part/package combination with a fixed location in the device (FPGA). input/output buffers, global clocks, and other chip-level resources are not inserted but can be instantiated within the module. The OOC is an important feature most useful in large hardware systems (designs), where the independent synthesis and implementation of the modules are necessary to extract the hardware properties. Modifying and reimplementing a large-scale system on a device takes hours or even days to complete by a synthesis tool, whereas a significantly shorter time for synthesis tool is needed for each system module owing to its smaller size.

[6]A digital hardware operates with pulses of an oscillator. The amount of time between pulses is referred to as a clock cycle or the inverse of clock frequency. For example, a hardware operating at a 100-MHz frequency has a clock cycle of 10 nanoseconds.

[7]A simulation that conforms to the cycle-by-cycle behavior of the target hardware.

$l_extendedprice \times (1 - l_discount)$, of the orders with the largest revenue of those that had not been shipped by a given date. Orders were listed in decreasing order of revenue. If there were more than ten unshipped orders, only ten orders with the largest revenue were listed. Further details are available in the TPC-H benchmark ^{TM} standard specification manual [30].

TPC-H Query Modification —

To execute the third query on our SSP, we needed to change the concept of the tables to sliding windows and feed each row in a table as a tuple to our stream processing engine. In queries with state operators, various sizes of the sliding window and the order of the reception of new tuples with respect to the origins of the streams affected the results. This is a property of stream processing engines that needs to be accounted for in practical use cases. Therefore, considering the effects of these factors is beyond the scope of this work.

To feed the rows of tables as tuples to SSP we allocate a sufficient number of bits per field used in the processing. In Figure 7.1.6, we show the mapping of the attributes of the third query table into different fields, where each attribute field is specified by a number of bits. Subsequently, the total width of a tuple for each stream was calculated by the sum of the bits of its fields.

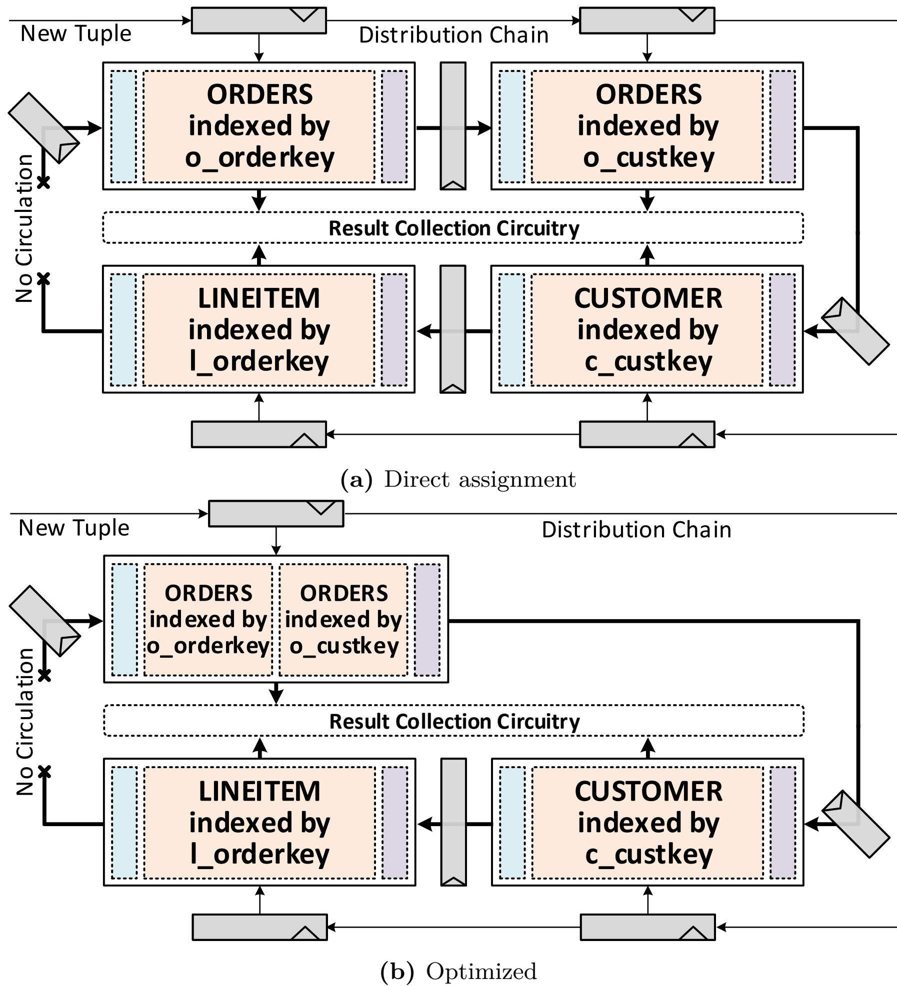
Query Assignment —

The choice of the topology of SSP is flexible and can be determined depending on the system requirements. In the following, we present an example of the TPC-H third query assignment to a SSP instance drawn in detail in Figure 7.1.7. The assigned components are color-coded to demonstrate the mapping between the VHDL implementation and the SSP topology.

The SSP instance for this example has four topology bricks arranged in four rows and one column. Note that there is no limit on the size and form of the chosen topology. In Figure 7.1.7a, the indexing values defined by ROW and COLUMN refer to each topology brick. For example, the first brick contains a bypass unit, a selection (Q3_SEL1) on the lineitem stream (built from its equivalent table), a selection (Q3_SEL2) on the customer stream, and a selection (Q3_SEL3) on the orders' stream.

A bypass unit is a pass-through without any internal component, and having one of these units in each topology brick is necessary to transfer instructions to the next row brick. The existence of the bypass unit prevents additional complications in the logic of other processing units by offloading the unrelated task of bypassing streams and instructions. Therefore, a topology brick can contain at least two units while the upper limit is defined by the application requirements and fan-out limitations of the chosen hardware.

The implementation of TOPO_MATRIX_INSTANCE in VHDL, Figure 7.1.7a, included the



(a) Direct assignment

(b) Optimized

Figure 7.3.1: Circular-MJ architecture customized for TPC-H 3rd query.

identifiers of processing units instantiated subsequently using the VHDL GENERATE construct, as shown in Figure 7.1.7b. As an example, the C_TPCH_Q3_CMJOIN_Inst hardware unit was instantiated and connected to the remaining components because its identifier (Q3_CMJOIN) existed, in Figure 7.1.7a.

From each tuple, only the necessary fields were fed into the hardware to avoid transmitting/handling unnecessary data. This part inherently implemented the projection operation as a task of the decomposer component (presented in Figure 7.2.5) in the complete system. The remaining operators were placed in the SSP instance, as shown in Figure 7.1.7c, with the exception of the aggregation and groupBy operators, which were merged together in one unit, Q3_GROUPBY_AGG, in Figure 7.1.7a.

To simplify the representation, we show the communication network (SCNoC) separate from the

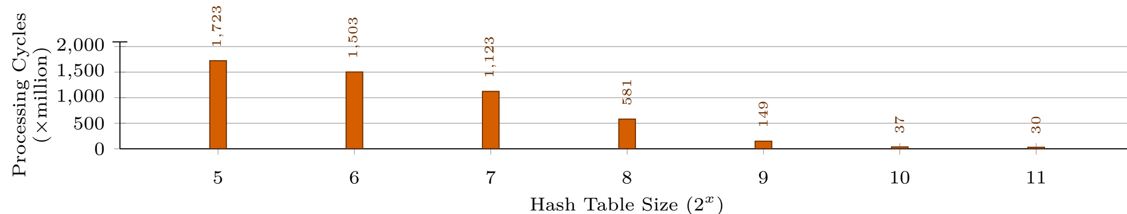


Figure 7.3.2: Input throughput vs. hash table size ($w : 2^{10}$).

SSP processing units in Figure 7.1.7d. The respective instructions for the programming of the SCNoC for the TPC-H third query are shown in Figure 7.1.7e. These instructions were fed into the SSP hardware one-by-one to define the paths of the streams in each GSwitch and LSwitch. In addition to the third query streams, we defined an extra stream that only contained one tuple to indicate the end of the streams, referred to as END_MESSAGE. The end tuple was fed into the SSP hardware after the end of all streams. The existence of this tuple was necessary for the groupBy and orderBy operators to notify them of the end of operations, pushing the resulting data from them.

Multiway Hash-based Stream Joins —

We use our hash-based join in a custom three-way stream join for the third query to show the potential throughput while making it possible to execute TPC-H benchmarks in reasonable time. Using our hash-based approach, we need to replicate the *orders'* stream stage, once indexed, based on the o_custkey field and once based on the o_orderkey field. The resulting architecture, the customized version of Circular-MJ (Figure 7.1.3), is shown in Figure 7.3.1a. Note that, due to the simplification of the multiway stream join operator in the third query, we do not need the circular path, which is shown by a disconnection mark in this figure. Because each incoming tuple was processed in, at most, one of the indexed sliding windows (either o_custkey or o_orderkey), we can further optimize the design by placing both the indexed order stream stages into one stage to improve the efficiency of the system, which leads to the architecture shown in Figure 7.3.1b.

In Figure 7.3.1b, each new tuple is stored in its corresponding sliding window. To process the orders' stream, tuples start execution at the customer stage, and the resulting tuples are emitted from the lineitem stage. The customer stream tuples start execution at the orders' stage, and the intermediate results of this stage pass through the customer stage without processing . The final results are output from the lineitem stage. Finally, tuples of the lineitem stream start processing at the orders' stage, and the resulting tuples are emitted from the customer stage.

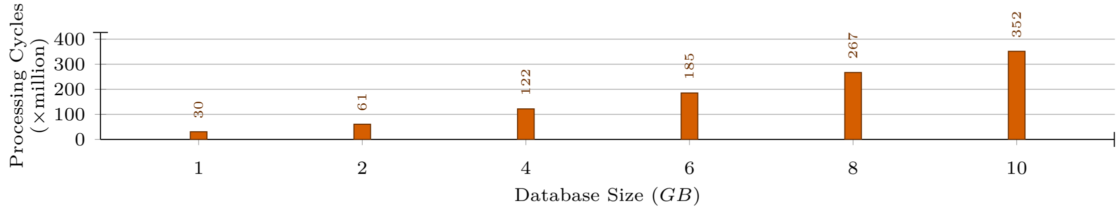


Figure 7.3.3: Input throughput vs. database size ($w : 2^{10}, ht : 2^{11}$).

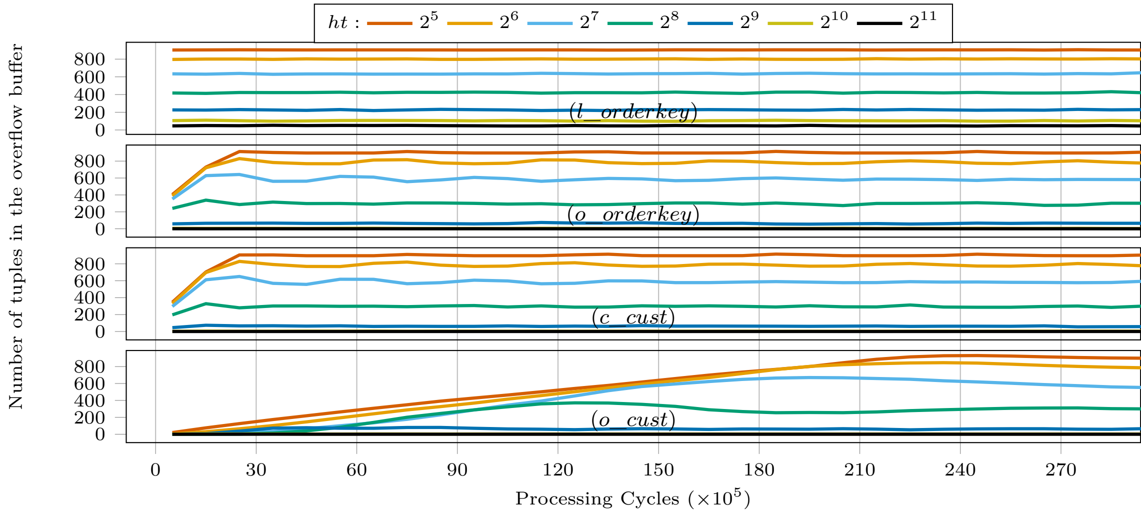


Figure 7.3.4: Number of tuples in the overflow buffer ($w : 2^{10}$).

7.3.2 Throughput Measurements

When working with database benchmarks (i.e. TPC-H), most join operators use equal conditions (equi-join), where using hashing techniques is the natural approach for acceleration. However, following acceleration using our proposed hash-based solution, the multiway equi-join operator remains the bottleneck in the TPC-H third query processing. The severity of this bottleneck is determined by the effectiveness of the hashing technique utilized.

The effect of the size of the hash table (ht) used in the optimized Circular-MJ in Figure 7.3.1b on the number of cycles needed to process all data streams is presented in Figure 7.3.2. The product of the cycle count and cycle period (i.e. 6.5 nanoseconds; extracted from the implementations, Figure 7.3.7) specifies the processing time. For example, with $ht = 2^9$, the entire query execution lasts 968.5 milliseconds ($(149 \times 10^6) \times (6.5 \times 10^{-9})$). As shown in this figure, an increase in the size of the hash tables, from left to right, improves processing throughput due to a reduction in the number of tuple insertion collisions in the hash tables. This significantly reduces the number of tuples in the overflow buffer (Figure 7.1.4), which uses a slow nested loop search.

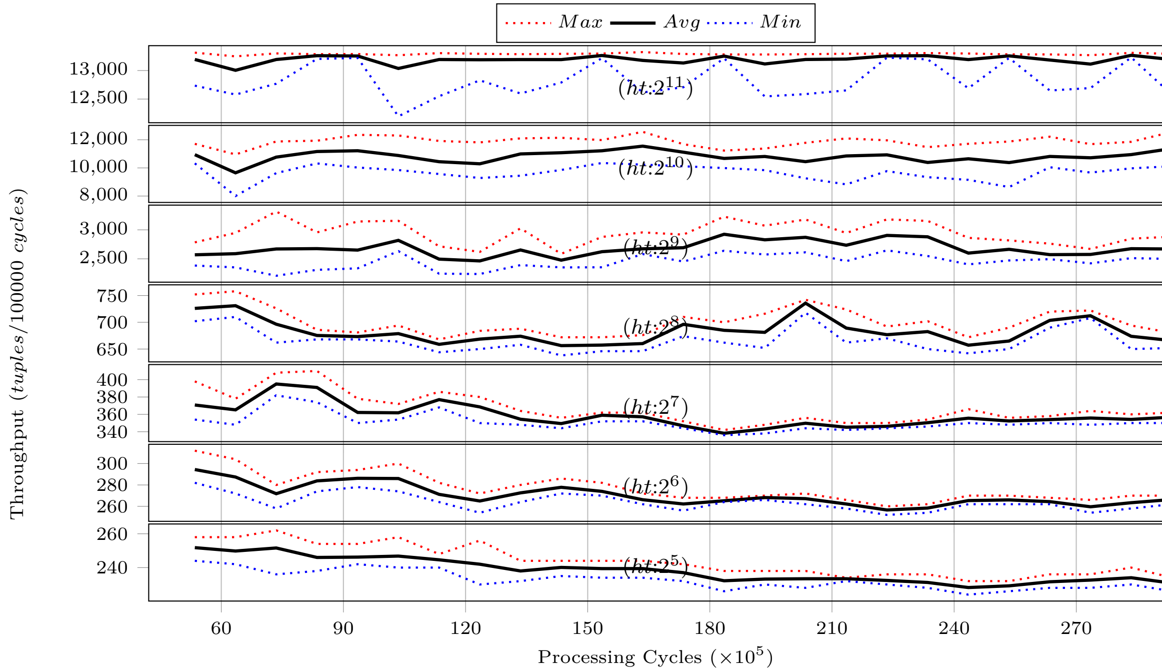


Figure 7.3.5: Input throughput measurements ($w : 2^{10}$).

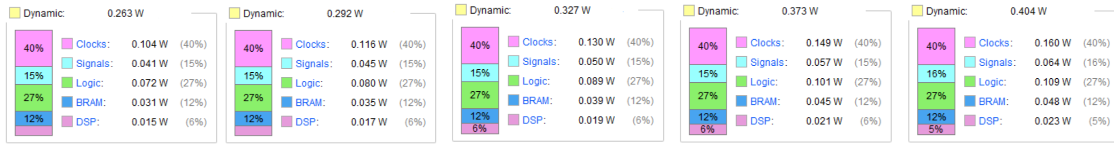
The effect of the size of the data stream (equivalent to the size of the database generated by the DBGen tool) on processing times for the TPC-H third query is illustrated in Figure 7.3.3. We observe a linear increase in processing time with the size of the data stream. With a clock period of 10 nanoseconds (clock frequency of 100 MHz), data streams of sizes one, two, four, six, eight, or 10 GB were processed in 300 ($30 \times (10 \times 10^{-9}) \times 10^6$), 610, 1220, 1850, 2670, or 3520 milliseconds, respectively.

The effectiveness of our hashing mechanism was determined by the number of tuples stored in the overflow buffer, as shown in Figure 7.3.4. A small number means fewer tuple insertion collisions and, therefore, more effective hashing, and vice versa. We show four sub-figures here, and each presents the effect on a single indexed attribute. This effectiveness was also influenced by the pattern of the received values and the size of each stream. In this figure, we observe a slower growth in the number of tuples in the overflow buffer for the order and customer streams compared with the lineitem stream; this result was obtained owing to the large difference in the sizes of these streams. In other words, we fed SSP with more tuples from the lineitem stream compared with the other two streams.

We observed a similar increase in the use of the overflow buffer as we reduced the size of the hash table for each stream. An interesting observation was the slow growth in the use of the overflow buffer for the o_cust attribute compared with the o_orderkey attribute (both from the orders' stream) due to the reception of more diverse values for the o_cust attribute with regard to the

Table 7.3.1: SSP resource utilization for TPC-H third query (Virtex UltraScale XCVU190-2FLGC2104E FPGA)

Unit	CLB	CLB LUTs	CLB Registers	CARRY8	F7 Muxes	F8 Muxes	LUT Logic	LUT Memory	LUT Flip Flop Pairs	Block RAM Tile	DSPs
Total	134280	107424	2148480	13420	537120	268560	1074240	231840	1074240	3780	1800
FQP-Q3	9221	20695	40985	251	3193	1580	15947	4748	4183	6	130
-Brick1	98	499	318	0	0	0	155	344	16	0	0
-Brick2	8281	16678	37858	240	2944	1456	13582	3096	3890	6	128
–Input_FIFO	13	68	4	0	0	0	12	56	2	0	0
–LSwitch	2	11	0	0	0	0	3	8	0	0	0
–CMJoin	8221	16197	37831	240	2944	1456	13445	2752	3872	0	128
—PipeStage1	2054	3963	9501	60	736	364	3301	662	946	1.5	32
—PipeStage2	4081	7867	18429	120	1472	728	6655	1212	1852	3	64
—PipeStage3	2081	4037	9477	60	736	364	3375	662	951	1.5	32
—Auxiliary	147	330	303	0	0	0	114	216	9	0	0
–Filters	46	240	16	0	0	0	16	224	8	0	0
–Collector	34	162	7	0	0	0	106	56	4	0	0
-Brick3	262	695	1074	9	121	60	631	64	75	0	2
-Brick4	461	1180	1690	2	128	64	836	344	152	0	0
-GSwitch1-5	∼5*52	∼5*275	5*9	0	0	0	∼5*151	5*124	∼5*5	0	0



(a) 10 nanoseconds. **(b)** 9 nanoseconds. **(c)** 8 nanoseconds. **(d)** 7 nanoseconds. **(e)** 6.5 nanoseconds.

Figure 7.3.6: Clock frequency period effect on the power consumption of various system parts.

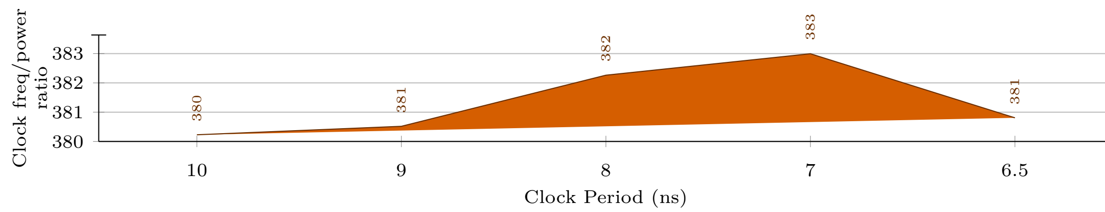


Figure 7.3.7: Working clock frequency vs. power consumption.

order of arrival of the tuples.

Figure 7.3.5 presents variations in the input throughput with the number of processing cycles for different hash table sizes ranging from 2^5 to 2^{11}. By reducing the effectiveness of the hashing mechanism, more tuples are stored in the overflow buffer. This translates into a more sequential comparison of tuples, which significantly reduces processing throughput. As a side effect, we can also clearly see the warm-up phase in the experiments with smaller hash tables, which shows longer processing times as a function of an increase in the number of tuples in the overflow buffer.

7.3.3 Evaluating the Implementation

A detailed summary of the hierarchical resource utilization for the third TPC-H query is presented in Table 7.3.1. The third TPC-H query is a particularly resource-intensive query due to its multiway stream join operator. However, the implementation on a state-of-the-art FPGA consumes only a fraction of the available resources. This demonstrates the applicability of SSP to deploy multiple queries with even more complex processing components.

In Table 7.3.1, we observe low use of block RAM tiles (six of 3,780) for hash tables and their corresponding components in the custom multiway stream join (Figure 7.3.1b). In this implementation, we used small sliding windows because the memory resources provided in an FPGA are valuable and limited. When working with large sliding windows, a decision concerning the use of one or multiple external memory chips (directly connected to the FPGA) needs to be made. This decision frees up valuable FPGA memory resources for logics and crucial buffering.

The TPC-H third query used five GSwitch-Ls (Figure 7.1.7), and they consumed a negligible amount of FPGA resources. Similarly, the LSwitchs consumed minimal resources from our FPGA, as observed in the resource utilization report of LSwitch in the second topology brick (`Brick_2`). Therefore, our SCNoC complied with its objectives of simple and undemanding communication components.

`Brick_1` contained the selection operators and consumed a negligible amount of resources, as expected of stateless operators, whereas `brick_2` consumed the most resources due to its multiway stream join operator. `Brick_3` contained the aggregation-groupBy operator and `brick_4` contained the order by operator; however, they also consumed a negligibly small amount of resources compared with the second topology brick.

In Table 7.3.1, we observe the resource consumption in the implementation of the pipeline stages for our custom multiway stream joins. *PipeStage2* used double the resources for the other two stages because it contained two separate instances of our HB-SJ, although sharing some components resulted in a slight reduction in terms of resource consumption.

7.3.4 Power Consumption Evaluations

Dynamic power consumption measurements, including the contributions of different types of FPGA resources, are presented in Figure 7.3.6. In these measurements, *clocks* shows the power consumed by the clock network, and *signals* shows the power consumed by the interconnections between components. *Logic* refers to the processing and routing components realized by FPGA configurable logic blocks (CLBs). *BRAM* shows the power consumed by memory elements in the design, and *DSP* presents the power consumed by the built-in (faster and more efficient) digital signal processing units, such as float multipliers.

In our case study, the processing throughput improved linearly in relation to the frequency of the operating clock. The ratios of processing throughput to power consumption for various working clock frequencies are presented in Figure 7.3.7. Although the difference in this ratio is better shown in ASIC solutions, owing to their higher clock frequencies and fixed implementation, we still observed a clear peak at a clock period of 7 nanoseconds ($\sim$142 MHz). In these implementations, we kept the other realization conditions intact to reduce the chance of changes in the implementation by using the synthesis tool due to its internal optimization. However, opting for a higher clock frequency (e.g. clock periods smaller than 6.5 nanoseconds) forced the synthesis tool to use more intrusive optimization techniques (i.e. replication of processing logic) to achieve the targeted clock frequency. In Figure 7.3.6, we observe a similar consumption for each category of components across multiple realizations, each with a different clock frequency. This implies that the synthesis tool generated similar implementations for various clock frequencies.

7.3.5 Implementing Stateless Queries

As expected of a hardware solution, the implementations of queries centered around stateless operators result in a very large processing throughput. This is the main reason for why we observed a first layer of hardware to filter incoming data before they were processed in the general-purpose processors in the computing centers.

The TPC-H first query (pricing summary report) is a good example of such queries. This query reports the volume of business that is billed, shipped, and returned. In this query, following the projection, lineitem tuples are fed into a selection operator, and the intermediate results are sent to an aggregation operator. At the end, the outcome of the aggregation operator is processed by a groupBy operator and then an orderBy operator. With only a single processing path, our SSP processed a database of size 1 GB in less than 22.4 million cycles, which translates into 224 milliseconds at a clock frequency of 100 MHz. Moreover, in such queries, it is possible to simply replicate the processing path multiple times to obtain a linear speedup.

Chapter 8

Conclusions

In this dissertation, we focused on hardware accelerated stream processing with the flexibility to adjust to query changes. This flexibility is essential to many streaming applications such as real-time data analytics, information filtering, and complex event processing.

Our proposed architecture tackle some well-known challenges, i.e., the memory-wall, power-wall, von Neumann bottleneck, and, overall, a staggering design complexity, using three mechanisms: (1) task-specialization and division of complex queries into small, self-contained, and independent components; (2) avoiding centralized and complex coordination units; and (3) increasing processor and memory coupling.

We used modular architectures in our designs while ensuring a distinct separation between the communication and processing components which are connected with well-defined interfaces. This greatly simplifies the addition of new components (e.g., query operators) or further developments on the architectures.

8.1 Summary

Our main goal has been to propose flexible hardware that can be used for various types of stream processing queries. Here we summarize our steps toward the flexible hardware accelerated stream processing in this dissertation.

In the first step in our research, we proposed the FQP which provides course- and fine-grained flexibility. The course-grained flexibility enables the selection of an appropriate topology for a set of queries using a set of predefined components that can be beneficial when utilizing reconfigurable hardware, i.e., FPGA. The fine-grained flexibility is made possible by the proposed programmable components (i.e., OP-Blocks and data routing switches) that enable the possibility of real-time updates on the processing procedure, provided these updates are supported by the programmable components.

The FQP had to use a bidirectional data-flow architecture as the only available method (at the time of its design) to support stream join parallelization. This choice imposed further complexities on query mapping (to FQP components). To address this issue, in the next step in our work we proposed SplitJoin, which benefited from a unidirectional data-flow architecture for the stream join parallelization.

To further tackle the stream join complexity, arguably as the most resource-intensive operation in stream processing, we proposed Circular-MJ to add the support for multiway stream join, as the next step in our research. Additionally, we designed and proposed some other important processing components, i.e., Stashed-MJ, to add a buffering capability to the stream join and HB-SJ, to support high-throughput stream processing for equi-join operations.

In the last step in this dissertation, we brought all of our concepts together in the design of our simplex stream processor (SSP) which benefits from our stream customized network-on-chip and the unidirectional data-flow architecture to build an elegant architecture.

We designed SSP with the modularity concept in mind. This modularity and the architecture simplicity, granted by the unidirectional data-flow, significantly improve two properties of the SSP 's predecessor (FQP). The first property is the support to add new components, that is improved due to the components' reduced number of input/output ports. This is important to motivate designers to build a rich open-source library, guaranteeing the wide-spread application of our solution. The other improved property is the query mapping which is now performed based on a straightforward component assignment on a topology with a linear arrangement of processing components.

8.2 Future Work

As common with hardware solutions, we need supplementary components (as previously shown in Figure 7.2.5) for our processor to readily utilize it in practice. In this regard, the following components are crucial to have a complete system for straightforward real-world integration:

i. **Decomposer/Composer:** to reduce the complexity of the main execution engine, we need to offload data manipulation operations to independent components. As common in data-intensive applications, we expect to have enormous data volumes at the input of our system. A large part of this data would not be interesting for the actual processing. Therefore, feeding the whole data volume to the executing engine of our system (SSP) is not practical. Also preprocessing of this large data volume using a general-purpose processor would be challenging due to the extreme memory and processing limitations (as already discussed in this dissertation). The common approach is to have a hardware processing layer to filter this data. As a result, we only need to feed the smaller but crucial data into the executing engine. This ensures that the processing components are fully utilized that is one of the main targets for efficient high-performance systems. Some parts of the data may not be necessary for the processing but still, they are important to be included in the processed data by the executed engine. To avoid feeding the unnecessary data to the executing engine we foresee the need for a decomposer component to separate different parts of the data and feed the right part to the executing engine and correspondingly a need for a composer component to add back the important data parts to the executing engine's resulting data.

ii. **SSP Designer:** our proposed stream processing architecture is based on a robust and modular definition of components and communication network interfaces. However, a system architecture still needs to arrange the right processing topology for its intended query. This arrangement is done by selecting the processing components from the provided library and placing them in the provided communication network architecture. Having a graphical user interface would make this arrangement as simple of drawing a block diagram. Ideally, we can have an automated design software which receives the queries and builds the right topology based on them.

iii. **Further Processing Components:** although we have provided the design for crucial operators, there are still other missing processing components. In our vision, we think of having an open-source community where individual designers can use from and contribute to the SSP library. This step would be essential for the wide-spread use of SSP and also motivates its further developments.

iv. **Query Assigner:** as two other important components in the complete system, we need to have a query assigner and a compiler to handle the query updates on the execution engine (SSP) in real-time. The support for size and type of these updates depends on the reprogramming properties of utilized processing components. Placing a selection component only allows for

updates on the filtering conditions while an OP-Block or even a general-purpose core (e.g., ARM core) allows for significant updates on the running queries without a need for a change in the topology.

List of Figures

1.0.1 Envisioned acceleration technology outlook. 2
1.0.2 General-purpose processor architecture vs. custom hardware architecture. . . . 3
1.0.3 Accelerator spectrum. 4

2.2.1 Sliding window concept in stream join. 18
2.2.2 Traditional stream join architecture. 18

4.1.1 OP-Block: I/O ports. 31
4.1.2 OP-Block: internal architecture. 32
4.1.3 Simultaneous tuple transmission issue. 34
4.1.4 Instruction and data format. 35
4.1.5 PU state diagrams. 35
4.1.6 (a) System bus (b) OP-Block. 35
4.1.7 SQL code example of a query. 36
4.1.8 Parallelism and pipelining arrangements. 37
4.2.1 SQL code example of a query. 39
4.2.2 Query assignment in OP-Chain topology. 40
4.2.3 Query assignment in partial parallel topology. 41
4.3.1 Stream processing element. 42
4.3.2 Result aggregation buffer (RAB). 42
4.3.3 Segment-at-a-time at entry to OP-Block. 43
4.3.4 Customer segregation query. 43
4.4.1 OP-Block Segment-at-a-time. 45
4.4.2 Segregation query. 45
4.5.1 Data stream reception and transmission stages. 46
4.5.2 Effect of window size on input throughput. 48
4.5.3 Maximum clock frequency of an OP-Chain on Virtex-5. 48
4.5.4 Power consumption report. 48
4.5.5 Synthesis times on Virtex-7 (XC7VX980T). 50

4.5.6 OP-Chain resource utilization on Virtex-5 (XC5VLX50T). 50
4.5.7 Effect of match probability on input (red) and output (blue) throughput. . . . 51
4.5.8 Partially parallel FQP topology. 52
4.5.9 Effect of tuple size on input tuple rate and input throughput. 53

5.1.1 Stream join data flow models (*JC* stands for join core): (a) bi-directional and (b) uni-directional (top-down). 56
5.1.2 SplitJoin concept. 57
5.1.3 SplitJoin storing and processing steps. 57
5.1.4 SplitJoin parallel architecture. 58
5.1.5 SplitJoin data distribution and processing example. 58
5.1.6 Punctuation in result gathering. 59
5.2.1 Punctuated resulting stream. 64
5.3.1 Low-latency handshake join overview [75]. 66
5.3.2 SplitJoin complete system. 68
5.4.1 Sliding window concept in stream join. 69
5.4.2 Traditional stream join architecture. 70
5.5.1 Bi-flow join core design. 70
5.5.2 Uni-flow join core design. 70
5.5.3 Uni-flow parallel stream join hardware architecture. 71
5.5.4 Storage core. 72
5.5.5 Processing core. 72
5.5.6 Throughput measurements for flow-based stream join on hardware. 73
5.5.7 Throughput measurements for flow-based stream join on software. 74
5.6.1 Throughput comparison. 75
5.6.2 Visiting latency comparison. 75
5.6.3 SplitJoin latency measurements. 77
5.6.4 SplitJoin processing pipeline. 78
5.6.5 Count-based SplitJoin throughput. 79
5.6.6 Selectivity effect on SplitJoin throughput (28 JCs). 80
5.6.7 Count-based SplitJoin latency reports (uniform distribution:$1 - 10^5$). 80
5.6.8 Ordering precision effect (28 JCs, uniform distribution:$1 - 10^4$). 80
5.6.9 Latency reports for uni-flow parallel stream join on hardware. 83
5.6.10 Latency reports for uni-flow design on software. 84
5.6.11 Uni-flow design clock frequencies on Virtex-5 and Virtex-7. 84

6.1.1 Reordering operators in multiway stream joins. 88
6.1.2 Multiway stream joins with circular design. 89
6.1.3 Intermediate result generation in a pipeline stage. 89
6.1.4 Circular-MJ architecture. 90
6.1.5 Order control unit architecture. 92

6.1.6 Join core lookup and store/expire controller state diagram. 94
6.1.7 Order unit controller state diagram. 94
6.2.1 Stashed-MJ architecture. 97
6.2.2 Intermediate result buffer. 98
6.2.3 Stash design and its internal components. 98
6.2.4 Stream count effect on input throughput ($w : 2^{14}$). 103
6.2.5 Match probability effect on throughput ($6 - way, w : 2^{14}$). 103
6.3.1 Stash behavior in the invalid state ($w : 2^{10}, s : 2^5, mp : 0.001\%$). 105
6.3.2 Stash with recomputation support. 106
6.4.1 Linear-MJ architecture. 108
6.4.2 Order control unit operation example. 112
6.5.1 Match probability effect on stage buffer usage ($1^{st} stage, 6 - way, w : 2^{14}$). . . . 114
6.5.2 Stage buffer usage ($3 - way, w : 2^{14}, mp : 0.1\%$). 115
6.5.3 Steady-state throughput measurements ($6 - way, w : 2^{14}, \sigma : 1$). 115
6.5.4 Origin arbitration using normal distribution and limiter. 116
6.5.5 Steady-state throughput measurements (tuples/s) ($6 - way, w : 2^{14}, mp : 0.01\%$). 116
6.5.6 Stage buffer max usage ($1^{st} stage, 6 - way, w : 2^{14}, \sigma : 1$). 116
6.5.7 Stage buffer average usage for all pipeline stages ($6 - way, w : 2^{14}, \sigma : 1, mp : 0.01$). 117
6.5.8 Steady-state throughput measurements with hash-based stream join ($w : 2^{14}, mp : 0.0001\%$). 118
6.5.9 Stash activity ($w : 2^{10}, s : 2^5, mp : 0.001\%$). 118
6.5.10 Stash utilization ($w : 2^{15}$). 118
6.5.11 Stash overwrite effect ($w : 2^{15}, s : 2^{10}, mp : 0.0001\%$). 118
6.5.12 a) Intermittent usage issue on a large stash. b) Tuples' origin stream effect when using a stash. c) Stash (disabled vs. enabled) effect on throughput. d & e) Match probability effect in Parallel-MJ throughput ($G - JCs : 16$(■)$, 32$(■)$, \& 64$(■)). . 120
6.5.13 G-JC count effect on throughput ($w : 2^{14}, mp : 0.001\%$). 122

7.1.1 Internal architecture of OP block. 128
7.1.2 Stream join dataflow models. 128
7.1.3 Scalable multiway stream joins (Circular-MJ). 130
7.1.4 Hash-based architecture for stream join (HB-SJ). 131
7.1.5 Aggregation-GroupBy unit. 132
7.1.6 Tuple-to-bit conversion. 133
7.1.7 TPC-H third query mapping on SSP. 133
7.2.1 SCNoC architecture. 134
7.2.2 SCNoC flow model. 134
7.2.3 SCNoC communication blocks. 135
7.2.4 GSwitch-L and LSwitch instruction sets. 139
7.2.5 SSP complete system. 140
7.2.6 Communication interface. 140

7.3.1 Circular-MJ architecture customized for TPC-H 3rd query. 143
7.3.2 Input throughput vs. hash table size ($w : 2^{10}$). 144
7.3.3 Input throughput vs. database size ($w : 2^{10}, ht : 2^{11}$). 145
7.3.4 Number of tuples in the overflow buffer ($w : 2^{10}$). 145
7.3.5 Input throughput measurements ($w : 2^{10}$). 146
7.3.6 Clock frequency period effect on the power consumption of various system parts. 148
7.3.7 Working clock frequency vs. power consumption. 148

List of Tables

4.5.1 OP-Chain synthesis parameters. 45
4.5.2 OP-Block resource utilization. 46
4.5.3 Virtex-5 (XC5VLX50T) specification. 47
4.5.4 Instruction latencies in OP-Block (operating clock frequency: 125MHz). 47
4.5.5 Tuple processing rate. 52

6.3.1 Pipeline stages access pattern (disabled stash). 104

7.3.1 SSP resource utilization for TPC-H third query (Virtex UltraScale XCVU190-2FLGC2104E FPGA) 147

List of Algorithms

1 SplitJoin distribution network. 61

2 A join core in SplitJoin. 62

3 Matches between **t** and sub-window **X**. 62

4 Expiring old tuples for time-based version. 62

5 Punctuation-based N-ary merger. 65

Bibliography

[1] Apache Hadoop. `http://hadoop.apache.org`. Accessed: 2019-05-21.

[2] Apache Storm. `http://storm.apache.org`. Accessed: 2019-05-21.

[3] *UG909: Vivado Design Suite User Guide: Partial Reconfiguration.* Accessed: 2019-05-21.

[4] Enhanced Intel SpeedStep technology for the Intel Pentium M processor white paper. *Retrieved from the web: ftp://download.intel.com/design/network/papers/30117401.pdf*, 2004.

[5] OpenPOWER cloud: Accelerating cloud computing. *IBM Research*, 2016.

[6] Social big data: Recent achievements and new challenges. *Information Fusion*, 2016.

[7] D. J. Abadi and Others. The design of the Borealis stream processing engine. CIDR, 2005.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous distributed systems. *OSDI*, page 44, 2016.

[9] I. Abaker, T. Hashem, I. Yaqoob, I. A. T. Hashem, N. Badrul Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan. The rise of "Big Data" on cloud computing: Review and open research issues. *Information Systems*, 2014.

[10] Z. A. Aghbari, I. Kamel, and T. Awad. On clustering large number of data streams. *Intelligent Data Analysis*, 16(1):69–91, 2012.

[11] N. Agrawal and A. Vulimiri. Low-latency analytics on colossal data streams with SummaryStore. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, pages 647–664, New York, NY, USA, 2017. ACM.

[12] K. Aingaran, S. Jairath, G. Konstadinidis, S. Leung, P. Loewenstein, C. McAllister, S. Phillips, Z. Radovic, R. Sivaramakrishnan, D. Smentek, and T. Wicki. M7: Oracle's next-generation Sparc processor. *IEEE Micro*, 35(2):36–45, 2015.

[13] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The Dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.

[14] M. H. Ali, W. G. Aref, R. Bose, A. K. Elmagarmid, A. Helal, I. Kamel, and M. F. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. *VLDB*, pages 1295–1298, 2005.

[15] 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*, pages 317–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[16] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. *Proceedings of the 21st International Conference on Data Engineering*, pages 118–129, 2005.

[17] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation*, NSDI, pages 1–14, 2004.

[18] R. R. Bordawekar and M. Sadoghi. Accelerating database workloads by software-hardware-system co-design. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, pages 1428–1431, May 2016.

[19] S. Breß, H. Funke, and J. Teubner. Robust query processing in co-processor-accelerated databases. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD*, pages 1891–1906, 2016.

[20] P. Carbone, S. Ewen, S. Haridi, A. Katsifodimos, V. Markl, and K. Tzoumas. Apache Flink: Unified stream and batch processing in a single engine. *Data Engineering*, 36:28–38, 2015.

[21] A. Caulfield, E. Chung, A. Putnam, H. Angepat, J. Fowers, S. Heil, J. Y. Kim, D. Lo, M. Papamichael, T. Massengill, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. *IEEE Micro*, pages 1–13, 2017.

[22] U. Çetintemel, D. Abadi, Y. Ahmad, H. Balakrishnan, M. Balazinska, M. Cherniack, J.-H. Hwang, S. Madden, A. Maskey, A. Rasin, E. Ryvkina, M. Stonebraker, N. Tatbul, Y. Xing, and S. Zdonik. *The Aurora and Borealis Stream Processing Engines*, pages 337–359. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016.

[23] 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. *CIDR*, 20(March):668, 2003.

[24] C. P. Chen and C.-Y. Zhang. Data-intensive applications, challenges, techniques and technologies: A survey on big data. *Information Sciences*, 2014.

[25] F. Chen, P. Deng, J. Wan, D. Zhang, A. V. Vasilakos, and X. Rong. Data mining for the internet of things: Literature review and challenges. *International Journal of Distributed Sensor Networks*, 2015(i):431047, 2015.

[26] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. *ACM SIGMOD Record*, 29(2):379–390, 2000.

[27] R. Chen and V. K. Prasanna. Accelerating equi-join on a CPU-FPGA heterogeneous platform. *Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM*, pages 212–219, 2016.

[28] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. *Proceedings of the Innovative Data Systems Research Conference*, pages 277–289, 2003.

[29] J. Cong and K. Minkovich. Optimality study of logic synthesis for LUT-based FPGAs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):230–239, feb 2007.

[30] T. P. P. Council. TPC-H benchmark specification. *Published at http://www. tcp. org/hspec. html*, pages 1–134, 2011.

[31] 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*, pages 647–651, 2003.

[32] M. Díaz, C. Martín, and B. Rubio. State-of-the-art, challenges, and open issues in the integration of Internet of things and cloud computing. *Journal of Network and Computer Applications*, 67:99–117, 2016.

[33] E. E. Drakonaki and G. M. Allen. Magnetic resonance imaging, ultrasound and real-time ultrasound elastography of the thigh muscles in congenital muscle dystrophy. *Skeletal Radiology*, 39(4):391–396, 2010.

[34] E. Fidler, H. a. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *In 8th International Conference on Feature Interactions in Telecommunications and Software Systems*, pages 12–30, 2005.

[35] M. Fontoura, S. Sadanandan, J. Shanmugasundaram, S. Vassilvitski, E. Vee, S. Venkatesan, and J. Zien. Efficiently evaluating complex boolean expressions. *Proceedings of the 2010 international conference on Management of data - SIGMOD*, page 3, 2010.

[36] P. Francisco. The Netezza data appliance architecture: A platform for high performance data warehousing and analytics, 2011.

[37] C. Franke. *Adaptivity in Data Stream Mining*. PhD thesis, University of California, Davis, 2008.

[38] M. Galkin, K. M. Endris, M. Acosta, D. Collarana, M.-E. Vidal, and S. Auer. Smjoin: A multi-way join operator for sparql queries. In *Proceedings of the 13th International Conference on Semantic Systems*, Semantics2017, pages 104–111, New York, NY, USA, 2017. ACM.

[39] B. Gedik, R. R. Bordawekar, and P. S. Yu. CellJoin: A parallel stream join operator for the cell processor. *VLDB Journal*, 18(2):501–519, 2009.

[40] B. Gedik, K. Wu, P. S. Yu, and L. Liu. GrubJoin: An adaptive, multi-way, windowed stream join with time correlation-aware CPU load shedding. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1363–1380, Oct 2007.

[41] T. Gomes, S. Pinto, T. Gomes, A. Tavares, and J. Cabral. Towards an FPGA-based edge device for the Internet of Things. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2015-October, 2015.

[42] X. Gu, P. S. Yu, and H. Wang. Adaptive load diffusion for multiway windowed stream joins. *ICDE*, 2007.

[43] V. Gulisano, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. ScaleJoin: a deterministic, disjoint-parallel and skew-resilient stream join. *IEEE Transactions on Big Data*, pages 1–1, 2016.

[44] A. Hagiescu, W. Wong, D. F. Bacon, and R. Rabbah. A computing origami: Folding streams in FPGAs. In *46th ACM/IEEE Design Automation Conference*, pages 282–287, July 2009.

[45] R. J. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer. Accelerating join operation for relational databases with FPGAs. *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, pages 17–20, 2013.

[46] M. A. Hammad, W. G. Aref, and A. K. Elmagarmid. Query processing of multi-way stream window joins. *VLDB Journal*, 17(3):469–488, 2008.

[47] 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 - International Conference on Data Engineering*, volume 20, page 851, 2004.

[48] T. Honjo and K. Oikawa. Hardware acceleration of Hadoop MapReduce. In *Proceedings - 2013 IEEE International Conference on Big Data, Big Data 2013*, pages 118–124, 2013.

[49] S. Idreos, E. Liarou, and M. Koubarakis. Continuous multi-way joins over distributed hash tables. *Proceedings of the International Conference on Extending Database Technology (EDBT)*, page 594, 2008.

[50] H. V. Jagadish, J. Gehrke, A. Labrinidis, Y. Papakonstantinou, J. M. Patel, R. Ramakrishnan, and C. Shahabi. Big data and its technical challenges. *Communications of the ACM*, 57(7):86–94, 2014.

[51] Y. Ji, J. Sun, A. Nica, Z. Jerzak, G. Hackenbroich, and C. Fetzer. Quality-driven disorder handling for m-way sliding window stream joins. *2016 IEEE 32nd International Conference on Data Engineering, ICDE 2016*, pages 493–504, 2016.

[52] X. Jin, B. W. Wah, X. Cheng, and Y. Wang. Significance and challenges of big data research. *Big Data Research*, 2(2):59–64, 2015.

[53] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings - International Conference on Data Engineering*, pages 341–352, 2003.

[54] T. Karnagel, D. Habich, B. Schlegel, and W. Lehner. The HELLS-join. *Proceedings of the Ninth International Workshop on Data Management on New Hardware - DaMoN '13*, page 1, 2013.

[55] C. R. Kitchin. *Telescopes and Techniques*, pages 1–264. Springer-Verlag New York, 2013.

[56] C. Kritikakis, G. Chrysos, A. Dollas, and D. N. Pnevmatikatos. An FPGA-based high-throughput stream join architecture. *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[57] 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*, pages 239–250. ACM, 2015.

[58] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM Computer Communication Review*, 36(4):339, 2006.

[59] T.-H. Kwon, H. G. Kim, M.-H. Kim, and J. H. Son. AMJoin: An advanced join algorithm for multiple data streams using a bit-vector hash table. *IEICE Transactions*, 92-D:1429–1434, 2009.

[60] Y. Law and C. Zaniolo. Load shedding for window joins on multiple data streams. In *2007 IEEE 23rd International Conference on Data Engineering Workshop*, pages 674–683, April 2007.

[61] J. W. Lockwood, A. Gupte, N. Mehta, M. Blott, T. English, and K. Vissers. A low-latency library in FPGA hardware for High-Frequency Trading (HFT). *Proceedings - 2012 IEEE 20th Annual Symposium on High-Performance Interconnects, HOTI*, pages 9–16, aug 2012.

[62] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system, 2003.

[63] R. Mueller, J. Teubner, and G. Alonso. Streams on wires — a query compiler for FPGAs. *Proc. of the VLDB Endowment*, 2(1):229–240, 2009.

[64] M. Najafi. The FQP project vision: Flexible query processing on a reconfigurable computing fabric. *SIGMOD*, 44(2):5–10, 2015.

[65] M. Najafi, M. Sadoghi, and H. Jacobsen. A scalable circular pipeline design for multi-way stream joins in hardware. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 1280–1283, April 2018.

[66] M. Najafi, M. Sadoghi, and H.-A. Jacobsen. Flexible query processor on FPGAs. *Proceedings of the VLDB Endowment*, 6(2):1310–1313, 2013.

[67] M. Najafi, M. Sadoghi, and H. A. Jacobsen. Configurable hardware-based streaming architecture using online programmable-blocks. In *Proceedings - International Conference on Data Engineering*, volume 2015-May of *ICDE'15*, pages 819–830, apr 2015.

[68] M. Najafi, M. Sadoghi, and H.-a. Jacobsen. SplitJoin : A scalable , low-latency stream join architecture with adjustable ordering precision. *USENIX Annual Technical Conference*, page 13, 2016.

[69] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga. Design and implementation of a handshake join architecture on FPGA. *IEICE Transactions on Information and Systems*, E95-D(12):2919–2927, 2012.

[70] D. Pascual Serrano, C. Vera Pasamontes, and R. Girón Moreno. Modelos animales de dolor neuropático. *Dolor*, 31(2):70–76, 2016.

[71] D. Pellerin. Announcing Amazon EC2 F1 instances with custom FPGAs hardware-accelerated computing on AWS, 2016.

[72] A. Putnam, G. Jan, G. Michael, H. S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. M. Caulfield, A. Smith, J. Thong, P. Yi, X. D. Burger, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. Prashanth, A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger. A reconfigurable fabric for accelerating large-scale datacenter services. *ACM SIGARCH Computer Architecture News*, 42(3):13–24, 2014.

[73] V. Raghavan, E. Rundensteiner, J. Woycheese, and A. Mukherji. FireStream: Sensor stream processing for monitoring fire spread. *ICDE*, 2007.

[74] M. E. Reid. MNS blood group system: A review. 25(3):95–101, 2009.

[75] P. Roy, J. Teubner, and R. Gemulla. Low-latency handshake join. *Proceedings of the VLDB Endowment*, 7(9):709–720, 2014.

[76] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H. A. Jacobsen. Multi-query stream processing on FPGAs. *Proceedings - International Conference on Data Engineering*, pages 1229–1232, apr 2012.

[77] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proceedings of the VLDB Endowment*, 3(1-2):1525–1528, 2010.

[78] M. Sadoghi, H. Singh, and H.-A. Jacobsen. fpga-ToPSS: Line-speed event processing on FPGAs. In *Proceedings of the 5th ACM International Conference on Distributed Event-based System*, DEBS '11, pages 373–374, New York, NY, USA, 2011. ACM.

[79] O. Segal, M. Margala, S. R. Chalamalasetti, and M. Wright. High level programming framework for FPGAs in the data center. *Conference Digest - 24th International Conference on Field Programmable Logic and Applications, FPL 2014*, 2014.

[80] R. P. S. Sidhu and V. K. Prasanna. Fast regular expression matching using FPGAs. In *FCCM*, pages 227–238, 2001.

[81] I. Sourdis and D. Pnevmatikatos. Fast, large-scale string match for a 10Gbps FPGA-based network intrusion detection system. In P. Y. K. Cheung and G. A. Constantinides, editors, *Field Programmable Logic and Application*, pages 880–889, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[82] J. Srivastava, D. and Golab, L. and Greer, R. and Johnson, T. and Seidel, J. and Shkapenyuk, V. and Spatscheck, O. and Yates. Enabling real time data analysis. *VLDB Endowment*, 3(1):1–2, sep 2010.

[83] Z. D. Stephens, S. Y. Lee, F. Faghri, R. H. Campbell, C. Zhai, M. J. Efron, R. Iyer, M. C. Schatz, S. Sinha, and G. E. Robinson. Big data: Astronomical or genomical? *PLoS Biology*, 13(7), 2015.

[84] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad. Database analytics acceleration using FPGAs. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, pages 411–420, 2012.

[85] J. Teubner and R. Müller. How soccer players would do stream joins. *Proc. ACM SIGMOD International Conference on Management of Data*, pages 625–636, 2011.

[86] J. Teubner and L. Woods. Data processing on FPGAs. *Synthesis Lectures on Data Management*, pages 1–118, 2013.

[87] 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. *SIGMOD*, 2014.

[88] A. Tumeo, O. Villa, and D. Sciuto. Efficient pattern matching on GPUs for intrusion detection systems. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF, pages 87–88, 2010.

[89] V. Turner, J. F. Gantz, D. Reinsel, and S. Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. *IDC Analyze the Future*, 16, 2014.

[90] T. Ueda, M. Ito, and M. Ohara. A dynamically reconfigurable equi-joiner on FPGA. *IBM Report*, 2015.

[91] T. Urhan and M. J. Franklin. XJoin: A reactively-scheduled pipelined join operator. *The Bulletin of the Technical Committee on Data Engineering*, 23(2):1–7, 2000.

[92] E. Valsomatzis and A. Gounaris. Driver input selection for main-memory multi-way joins. *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 818–825, 2013.

[93] E. Van Heerden, A. Karastergiou, S. J. Roberts, and O. Smirnov. New approaches for the real-time detection of binary pulsars with the square kilometre array (SKA). In *2014 31th URSI General Assembly and Scientific Symposium, URSI GASS 2014*, 2014.

[94] S. Venkataraman, A. Panda, K. Ousterhout, M. Armbrust, A. Ghodsi, M. J. Franklin, B. Recht, and I. Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, pages 374–389, New York, NY, USA, 2017. ACM.

[95] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, pages 285–296, 2003.

[96] C. Wang, X. Li, and X. Zhou. SODA: software defined fpga based accelerators for big data. *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 884–884, 2015.

[97] S. Wang and E. Rundensteiner. Scalable stream join processing with expensive predicates: Workload distribution and adaptation by time-slicing. *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 299–310, 2009.

[98] Z. Wang, J. Paul, H. Y. Cheah, B. He, and W. Zhang. Relational query processing on OpenCL-based FPGAs. *FPL 2016 - 26th International Conference on Field-Programmable Logic and Applications*, 2016.

[99] L. Woods, J. Teubner, and G. Alonso. Complex event detection at wire speed with FPGAs. *Proc. VLDB Endow.*, 3(1-2):660–669, 2010.

[100] L. Wu, R. J. Barker, M. A. Kim, and K. A. Ross. Navigating big data with high-throughput, energy-efficient data partitioning. *SIGARCH Comput. Archit. News*, 41(3):249–260, June 2013.

[101] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross. Q100: The architecture and design of a database processing unit. *SIGARCH Comput. Archit. News*, 42(1):255–268, 2014.

[102] J. Xie and J. Yang. A survey of join processing in data streams. *Data Streams: Advances In Database*, pages 209–236, 2007.

[103] J. Yang and J. Goodman. Symmetric key cryptography on modern graphics hardware. In *Lecture Notes in Computer Science*, volume 4833, page 249. Advanced Micro Devices, Inc., 2007.

[104] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. *HotCloud*, pages 10–10, 2010.

[105] 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, pages 423–438, New York, NY, USA, 2013. ACM.

[106] D. Zhang, J. Li, K. Kimeli, and W. Wang;. Slidingwindow based multi-join algorithms over distributed data streams. *Proceedings of the 22nd International Conference on Data Engineering - ICDE*, page 139, 2006.

[107] Y. Zhang, R. Chen, and H. Chen. Sub-millisecond stateful stream querying over fast-evolving linked data. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, pages 614–630, New York, NY, USA, 2017. ACM.

[108] Y. Zhou, Y. Yan, F. Yu, and A. Zhou. PMJoin: Optimizing distributed multi-way stream joins by stream partitioning. *Database Systems for Advanced Applications (DASFAA)*, 3882:325–341, 2006.

[109] S. Zhu, G. Fiameni, G. Simonini, and S. Bergamaschi. SOPJ: A scalable online provenance join for data integration. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 79–85, July 2017.

[110] X. Zhu, H. Gupta, and B. Tang. Join of multiple data streams in sensor networks. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1722–1736, 2009.