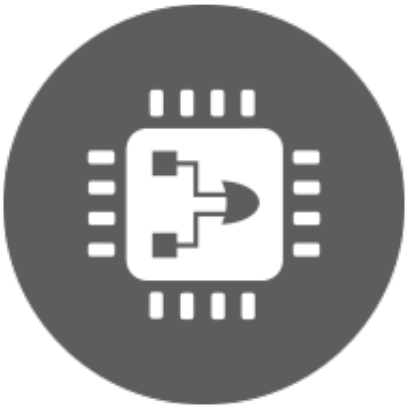
Πολυτεχνειο Κρητης

Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων

Παράλληλη Αρχιτεκτονική για Υλοποίηση του Αλγόριθμου Scalejoin στον Υπερυπολογιστή Convey

Διπλωματικη Εργασια

Καρανδεινός Χ. Έκτωρ



Εξεταστική Επιτροπή:

Καθ. Δόλλας Απόστολος(Επιβλέπων)

Αναπλ. Καθ. Ιωάννης Παπαευσταθίου

Καθ. Γαροφαλάκης Μίνως

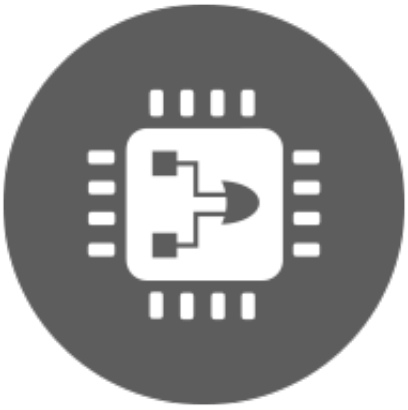
Χανιά, Ιούνιος 2017

Technical University of Crete
Department of Electrical and Computer Engineering

Parallel Architecture for the Scalejoin Algorithm Implementation on the Convey Supercomputer

Diploma Thesis

Karandinos C. Ektor



Thesis Committee:

Prof. Dollas Apostolos(Supervisor)

Assoc. Prof. Ioannis Papaefstathiou

Prof. Garofalakis Minos

Chania, June 2017

Abstract

Stream join consists one of the most resource-intensive operators in stream processing. Due to this characteristic, a big interest has been created in building high throughput and low latency systems which will be able to process real-time, bursty and rate varying data streams.

This thesis proposes an FPGA-based architecture which is based on one of the most efficient stream join algorithms, i.e ScaleJoin. The proposed architecture extends the first hardware-based architecture of the ScaleJoin algorithm. The first hardware implementation achieves high throughput and scalability but suffers from low resources utilization. In this thesis, we propose a novel architecture to achieve greater level of parallelism and exploit the available resources. Our system runs on Convey HC-2ex hybrid computer equipped with two six-core Intel Xeon E5-2640 processors running at 2.5 GHz and four Virtex 6 LX760 FPGAs. The experimental performance evaluation shows that our system totally outperforms the corresponding software-based solution and improves greatly the performance of the first hardware implementation.

Keywords

Stream processing, stream join, ScaleJoin, stream join operator, FPGA architecture, reconfigurable logic

στους γονείς μου

Ευχαριστίες

Θα ήθελα καταρχήν να ευχαριστήσω τον καθηγητή και επιβλέπων της παρούσας διπλωματικής κ. Απόστολο Δόλλα για το ενδιαφέρον που έδειξε από την πρώτη στιγμή, καθώς επίσης και τον αναπληρωτή καθηγητή κ. Ιωάννη Παπαευσταθίου και τον καθηγητή κ. Μίνω Γαροφαλάκη για την ευκαιρία που μου έδωσαν να ασχοληθώ με αυτό που ήθελα. Επίσης θα ήθελα να ευχαριστήσω ιδιαίτερα τον Γρηγόρη Χρυσό αλλά και τον Χρήστο Ρουσόπουλο για την πολύτιμη καθοδήγηση τους και τις ώρες που αφιέρωσαν για να με βοηθήσουν. Τέλος ευχαριστώ τους γονείς μου για τη στήριξη που μου έδειξαν καθ' όλη τη διάρκεια της φοίτησης μου αλλά και όλους τους κοντινούς μου ανθρώπους που μου στάθηκαν όλο αυτό το διάστημα.

Χανιά, Ιούνιος 2017

Καρανδεινός Χ. Έκτωρ

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter explains the main concepts of stream data mining in order to introduce the reader to the objectives of this thesis. Also, the terminology used in the following chapters is explained and analysed here. Last, this chapter describes the contribution of this thesis.

The chapter is organized as follows. In section 1.1 the basic terms used in the following chapters are explained, so that the reader becomes familiar with the concepts of this thesis' field. Also, an introduction to the field of stream data mining and more specifically to the stream join processing is made. Section 1.2 presents the contribution of this work. Finally, the structure of this thesis is analysed in section 1.3.

1.1 Terminology

Stream

In computer science, a stream is a sequence of data elements made available over time. A stream can be thought as items on a conveyor belt that are being processed one at a time rather than in large batches. Streams are processed differently from batch data since normal functions cannot operate on streams as a whole, as they have potentially unlimited data. In our case, we are concerned with stream processing, which is the continuous flow of data that is processed in a dataflow programming language, as soon as the program state meets the starting condition of the stream.

Tuple

A tuple is a data structure that consists of an ordered list of elements. Tuples are often used to describe other mathematical objects, such as vectors. In computer science, tuples

are directly implemented as product types in most functional programming languages. More commonly, they are implemented as record types, where the components are labeled instead of being identified by position alone.

Big Data

Big data is a term for data sets that are so large or complex that traditional data processing applications are inadequate to deal with them. Challenges include analysis, capture, data curation, search, sharing, storage, transfer, visualization, querying, updating and information privacy. The term "big data" often refers simply to the use of predictive analytics, user behavior analytics, or certain other advanced data analytics methods that extract value from data, and seldom to a particular size of data set. Big data, usually includes data sets with sizes beyond the ability of commonly used software tools to capture, curate, manage and process data within a tolerable elapsed time. Big data "size" is a constantly moving target, as of 2012 ranging from a few dozen terabytes to many petabytes of data. Big data requires a set of techniques and technologies with new forms of integration to reveal insights from datasets that are diverse, complex, and of a massive scale. Finally, big data can be described by the following three characteristics:

Volume: The volume of data implies scaling the storage and being able to perform distributed querying for processing. Solutions for the volume problem are either the use of data warehousing techniques or use parallel processing architecture systems.

Velocity: Velocity deals with the rate in which data is generated and flows into a system. Everyday applications generate unbounded amount of information that can be used in many ways for predictive purposes and analysis. Velocity not only deals with the rate of data generation but also with the speed in which an analysis can be returned from this generated data. Having real-time feedback is crucial when dealing with fast evolving information such as stock markets, social networks, sensor networks, mobile information and many others.

Variety: One problem in big data is the variety of data representations. Data can have many different formats depending on the source's nature, therefore dealing with this variety of formats can be challenging. Distributed key value stores, commonly referred as NoSQL databases, come in very handy for dealing with variety due to the unstructured way of storing data. This flexibility provides an advantage when dealing with big data. Traditional relational databases would imply restructuring

the schemes and remodeling when new formats of data appear.

Stream Data Mining

Data Stream Mining is the process of extracting knowledge structures from continuous, rapid data records. Examples of data streams include computer network traffic, phone conversations, ATM transactions, web searches, and sensor data. Data stream mining can be considered a sub-field of data mining, machine learning, and knowledge discovery.

In many data stream mining applications, the goal is to predict the class or value of new instances in the data stream given some knowledge about the class membership or values of previous instances in the data stream. Machine learning techniques can be used to learn this prediction task from labeled examples in an automated fashion. In many applications, especially operating within non-stationary environments, the distribution underlying the instances or the rules underlying their labeling may change over time, i.e. the goal of the prediction, the class to be predicted or the target value to be predicted. This problem is referred as concept drift.

As one can easily understand, processing Stream and Data Mining algorithms combination cannot simply "work" to get Stream Data Mining. Many Data Mining algorithms are used to process streams but they need to match new challenges and make adjustments. An example of these challenges and adjustments is the input of the algorithm, which in streams is infinite in constrast with Data Mining, where the data is finite and the configured algorithm must have a real-time response. In Chapter 2 a more analytical reference is made to the problems that Stream Data Mining algorithms have and the challenges of each type of algorithm.

A fundamental operator for the data stream mining is the stream join operator. Stream join is used to correlate the information from different streams. The join operation, usually takes place over specific time-based windows due to the unbounded size of the data streams. The stream join operator is computationally expensive and there are many works that focus on accelerating its processing using distributed or parallel frameworks. There exist published works on how to accelerate stream join processing using multicore platforms [23][45][41][34] and other works that use hardware-based solutions [35][29][21]. Thus, the ScaleJoin algorithm [23] is a new, parallel formulation of the stream join operator that uses a shared-memory framework. The algorithm offers high performance results, outperforming any other state-of-the-art stream join implementation. The main advantages of the ScaleJoin algorithm is that it can process tuples coming from an arbitrary number of in-

put streams and it guarantees deterministic processing with scalable and high-throughput parallelism. ScaleJoin algorithm is presented and analysed in Chapter 3.

1.2 Contribution

This section describes the contributions of this thesis. The current work presents a hardware-based architecture for one of the state-of-the-art stream join algorithms, i.e. ScaleJoin.The proposed architecture achieves high throughput and outperforms any other software-based and hardware-based solutions on this problem. More analytically the contributions of our work are presented below.

- The current work improves the performance of the first hardware-based implementation for the ScaleJoin algorithm by proposing a new hardware architecture with a larger number of processing units (PUs) and greater parallelism.
- The proposed architecture outperforms any other previous, either software or hardware-based, solutions for the stream join problem. More specifically our system carries out 38 times more comparisons per sec than the software-based solution, which runs on a high-end 48 core multiprocessor platform. The proposed system, also, manages to outperform the first hardware-based implementation of the ScaleJoin algorithm by two times as far as the comparisons rate per sec.
- The proposed hardware system offers greater throughput data rates than the previous proposed systems. In more details, the new system achieves at least 6 times better throughput data rate vs. the software solution and 1.4 times better rate vs. the first hardware-based solution.
- Our architecture exploits the full bandwidth of the Convey's HC-2ex memory and it almost doubles the utilization of the available resources.
- The proposed hardware-based architecture is scalable and generic, which means that it can be used for many different streaming problems that need to correlate streaming data.

1.3 Thesis Structure

The rest of this thesis is organized as follows. Chapter 2 presents algorithms and platforms regarding stream mining and makes an introduction to the stream join problem. In addition some of the most important implementations (software and hardware) in stream join processing are presented and analysed. Chapter 3 describes the ScaleJoin algorithm and its first hardware implementation. We address the improvements than can be made and make a first plan for our design. Chapter 4 presents and analyses our proposed architecture while in Chapter 5 takes place the evaluation of our work. Finally chapter 6 draws the conclusion of this work.

Chapter 2

Related Work

Data streams are infinite and high speed sequences of data instances. Mining of these large scale data streams to perform some kind of machine learning or futuristic predictions regarding data instances have drawn a significant attention of researchers in couple of previous years. The data streams resemble the real time incoming data sequence very well. The source of these data streams can be various sensors situated in medical domain to monitor health conditions of patients, in industrial domain to monitor manufactured products, etc. Other sources are user web click streams on social networking, e-commerce sites etc, twitter posts, various blogs, web logs, and many more [20] [8]. The above mentioned sources not only produce data streams, but they produce them in huge amount (of scale of tera bytes to peta bytes) and at rapid speed. Now, mining such huge data in real time raises various challenges and has become the hot area of research recently. These challenges include memory limitation and faster computing requirement.

The data stream mining task can be considered the same as traditional data mining task in terms of objective but quite different in terms of processing or executing the mining task. The reason behind this difference is the underlying challenges of infinite high speed data streams. This task, makes the traditional data mining algorithms and techniques incapable of appropriately handling data streams and yields the requirement of algorithms suitable for streaming data mining. This may be achieved in two ways; either modify the existing data mining algorithms to make them suitable for stream mining or create new streaming data mining algorithms right from the scratch. Something equally important data stream mining requires, is new platforms for computing and mining large scale data streams in real time. These platforms are required for various purposes such as data summarization, data streams aggregation from multiple sources, facilitating APIs for developing streaming data mining algorithms, etc. [17] [10] [7] [47] [43].

In this chapter, we address some of the main stream data mining algorithms and platforms in Sections 2.1 and 2.2. In Section 2.3, we point out the significance and the particularity of stream join processing. Finally, Sections 2.4 and 2.5 present some implementations of stream join algorithms implemented in software and hardware respectively.

2.1 Stream Mining Algorithms

2.1.1 Data Stream Clustering

Clustering is a widely studied problem in the data mining literature. However, it is more difficult to adapt arbitrary clustering algorithms to data streams because of one-pass constraints on the data set. An interesting adaptation of the k-means algorithm has been discussed in [42], which uses a partitioning based approach on the entire data set. This approach uses an adaptation of a k-means technique in order to create clusters over the entire data stream. In the context of data streams, it may be more desirable to determine clusters in specific user defined horizons rather than on the entire data set. A technique worth mentioning is the micro-clustering technique [12], which determines clusters over the entire data set. There is also a variety of applications of micro-clustering, which can perform effective summarization based analysis of the data set. For example, micro-clustering can be extended to the problem of classification on data streams [13]. In many cases, it can also be used for arbitrary data mining applications such as privacy preserving data mining or query estimation.

2.1.2 Data Stream Classification

The problem of classification is perhaps one of the most widely studied in the context of data stream mining. The problem of classification is made more difficult by the evolution of the underlying data stream. Therefore, effective algorithms need to be designed in order to take temporal locality into account. A wide variety of data stream classification algorithms exist, some of them are designed to be purely one-pass adaptations of conventional classification algorithms [36], whereas others (such as the methods in [13] and [19]) are more effective in accounting for the evolution of the underlying data stream. Classic example of Stream Classification algorithms is Very Fast Decision Trees algorithm (VFDT) or Hoeffding Trees developed by Domingos and Hulten [15] and it is also used in Samoa. This algorithm splits the tree using the current best attribute taking into account that the number of examples used satisfies the Hoeffding bound. VFDT is an extended version of

such a method which can address the research issues of data streams. Another algorithm worth mentioning is the on Demand Classification. Aggarwal et al. have adopted the idea of micro-clusters introduced in CluStream [27] in On-Demand classification in [9]. The on-demand classification method divides the classification approach into two components. One component constantly stores summarized statistics about the data streams and the second one, continuously, uses the summary statistics to perform the classification. The summary statistics are represented in the form of class-label specific micro-clusters. This means that each micro-cluster is associated with a specific class label, which defines the class label of the points in it.

2.1.3 Frequent Pattern Mining

The problem of frequent pattern mining was first introduced in [38], and was extensively analyzed for the conventional case of disk resident data sets. In the case of data streams, one may wish to find the frequent itemsets either over a sliding window or the entire data stream [14] [39]. Frequent patterns can not only effectively summarize the underlying datasets, providing key sights into the data, but also serve as the basic tool for many other data mining tasks, including association rule mining, classification, clustering, and change detection among others [26] [25] [28]. Many efficient frequent pattern algorithms have been developed in the last decade [22] [24] [30]. These algorithms typically require datasets to be stored in persistent storage and involve two or more passes over the dataset. Recently, there has been much interest in data arriving in the form of continuous and infinite data streams. In a streaming environment, a mining algorithm must take only a single pass over the data. Such algorithms can only guarantee an approximate result.

2.1.4 Change Detection in Data Streams

An important problem in the field of data stream analysis is change detection and monitoring. In many cases, the data stream can show changes over time, which can be used for understanding the nature of several applications. In many cases, it is desirable to track and analyze the nature of these changes over time. In [44] [16] [11], a number of methods have been discussed for change detection of data streams. The presence of evolution in data streams may also change the underlying data to the extent that the underlying data mining models may need to be modified to account for the change in data distribution.

2.1.5 Stream Cube Analysis of Multi-dimensional Streams

Much of stream data resides at a multi-dimensional space and at rather low level of abstraction, whereas most analysts are interested in relatively high-level dynamic changes in some combination of dimensions. To discover high-level dynamic and evolving characteristics, one may need to perform multi-level, multi-dimensional on-line analytical processing (OLAP) of stream data. Such necessity calls for the investigation of new architectures that may facilitate on-line analytical processing of multi-dimensional stream data [46] [18].

2.1.6 Sliding Window Computations in Data Streams

Many of the synopsis structures which are discussed, use the entire data stream in order to construct the corresponding synopsis structure. The sliding-window model of computation is motivated by the assumption that it is more important to use recent data in data stream computation [32]. Therefore, the processing and the analysis is only done on a fixed history of the data stream. To be more specific data elements arrive at every instant; each data element expires after exactly N time steps; and, the portion of data that is relevant to gathering statistics or answering queries is the set of last N elements to arrive. The sliding window refers to the window of active data elements at a given time instant and window size refers to N.

2.2 Stream Mining Platforms

Stream mining platforms are the frameworks that facilitate creation or collection of data streams as well as integration of various algorithms and APIs of stream mining to enable a developer or user to easily mine the data streams and evaluate the results. List of various data stream mining platforms along with their main focus are listed in Table 1. The stream processing frameworks listed in Table 2.1 can be categorized into three basic units. First unit includes stream preprocessing frameworks [9-12,57-58] that perform collection, filtering, aggregation and integration over data streams. Second unit of frameworks include stream processing engines that facilitate libraries and APIs, which provide faster manipulation of data streams such as S4 [31], Storm [4], Spark [47], Samza [2], etc. and facilitate streaming data mining libraries such as MOA [6], Spark [47], SAMOA [37], etc. Third unit of frameworks usually focus on analytical processing such as SQLstream Blaze [1], Pulser [33], etc. The distributed processing frameworks, have received a lot of attention from research and industries. These frameworks can handle huge scale of streaming data

computation and analytics.

Stream Mining Platform	Year	Major Focus
Aurora [17]	2003	Faster computing of data streams (from multiple sources) as defined by application administrator.
Scribe [10]	2004	Real time aggregation of streaming data from various sources
Borealis [7]	2005	Used for faster processing of incoming data streams.
Vowpal-Wabbit [3]	2007	Scalable, fast computing and integration of variety of data.
MOA [6]	2010	Performs low scale data stream computing. A GUI based framework that contains bulk of streaming data mining algorithms.
Apache S4 [31]	2010	Distributed faster data stream processing engine.
Apache Spark [47]	2010	Provides in memory data stream computation platform on Hadoop data stores. It is 10 times faster than MapReduce computation paradigm of Hadoop.
Storm [4]	2011	Distributed faster data stream processing engine.
Samza [2]	2013	A fault tolerant and scalable distributed framework for data stream processing.
SAMOA [37] [40]	2013	Provides large scale data stream computation through distributed framework. It can be easily integrated with other stream processing engines such as Storm and S4. Also, it has library of distributed mining algorithms for streams of data.
Amazon Kinesis [43]	2013	A cloud based service that provides real time distributed processing of large scale data streams. It can potentially capture terabytes data per hour, coming from thousands of sources such as financial transactions, web lick streams, social media, etc.
streamDM [5]	2014	An open source framework that collaborates with apache spark and is effective in mining big scale data streams.
Kafka Integrated SQLstream Blaze [1]	2014	Provides high performance distributed processing of data streams via SQLstream Blaze stream processing suite, for real time aggregation, analysis and visualization of large scale data streams.
Pulser [33]	2015	Open source framework for capable of capturing and processing large scale (around million) events and analytics in seconds. It can create custom data streams in order to perform real time business activity monitoring and reporting, fraud detection etc.

Table 2.1: *Stream Mining Platforms*

The stream processing frameworks listed in Table 2.1 can be categorized into three basic units. First unit includes stream preprocessing frameworks [9-12,57-58] that perform collection, filtering, aggregation and integration over data streams. Second unit of frameworks include stream processing engines that facilitate libraries and APIs, which provide faster manipulation of data streams such as S4 [31], Storm [4], Spark [47], Samza [2], etc. and facilitate streaming data mining libraries such as MOA [6], Spark [47], SAMOA [37], etc. Third unit of frameworks usually focus on analytical processing such as SQLstream Blaze [1], Pulser [33], etc. The distributed processing frameworks, have received a lot of at-

tention from research and industries. These frameworks can handle huge scale of streaming data computation and analytics.

2.3 Data Stream Join Processing

Given the fundamental role that is played by joins in querying relational databases, it is not surprising that stream join has also been the focus of much research on streams. Recall that relational (theta) join between two non-streaming relations R1 and R2, denoted $R1 \bowtie R2$, returns the set of all pairs $\langle r1, r2 \rangle$, where $r1 \in R1$, $r2 \in R2$, and the join condition $\theta(r1, r2)$ evaluates to true. A straightforward extension of join to streams gives the following semantics (in rough terms): At any time t, the set of output tuples generated thus far by the join between two streams S1 and S2 should be the same as the result of the relational (nonstreaming) join between the sets of input tuples that have arrived thus far in S1 and S2.

Stream join is a fundamental operation for relating information from different streams. This is especially useful in many applications such as sensor networks, where the streams that arrive from different sources may need to be related with one another. Due to the high computational cost of stream joins, a big interest has been created in achieving high throughput and low latency. For this reason, many implementations have been proposed both in software and hardware. The main goal of these implementations is to parallelize the processing of stream joins. Most of such works use multi-core CPUs, field programmable gate arrays (FPGAs) or massively parallel processor arrays (MPPAs). Some of the above implementations are presented below.

2.3.1 Software Based Implementations

CellJoin [21]

CellJoin is an algorithm that follows a three-step procedure for stream join, as described by Kang *et al.*[cytation here]. CellJoin algorithm is mapped on a cell processor, which is an heterogeneous multi-core architecture. The algorithm parallelizes the scan task over the available processing units achieving high performance. The main idea of the algorithm is based on the assignment of each in-memory window partition of one stream to the available CPU cores so that the scan is parallelized. Although CellJoin achieves low latency, its scalability is very limited due to the re-partition needed every time a new tuple arrives. The overhead of window re-partitioning grows linearly with the core number and the input

stream rate causing CellJoin algorithm to scale poorly.

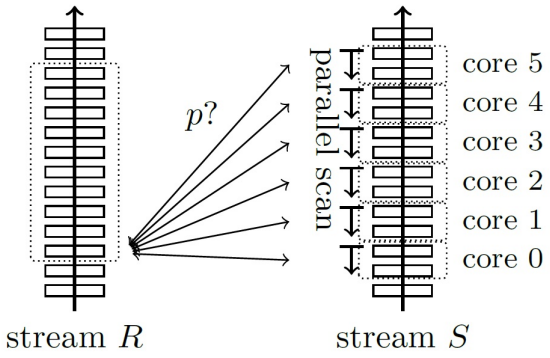


Figure 2.1: *CellJoin*

Handshake Join [45]

Handshake Join algorithm main goal is to scale out to very high degrees of parallelism in order to avoid the bottleneck that occurs with traditional approaches (CellJoin). To understand how Handshake Join works we can resemble it with soccer players. To be more specific, as soccer players, before the begging of every match shake hands, by walking in opposite directions, Handshake Join compare both streams, flowing in opposite directions, in parallel. This implementation is scalable enabling the use of multiple cores to increase the window size without limiting performance. Handshake Join replaced the classical three-step procedure, which was used by the stream join implementations.

Low Latency Handshake Join [41]

Low Latency Handshake Join algorithm, aims to maintain the positive characteristics of the original algorithm (scalability and high throughput). At the same time, it attempts to reduce latency and produce a deterministically ordered output. The high latency of the original algorithm is caused due to the queuing of tuples along the distributed windows. To avoid that bottleneck Low Latency Handshake Join forwards newly incoming tuples and stores them into a node-local window on the core that they belong. This way, each incoming tuple is "seen" by all involved processing units, it is stored in the respected node-local window and then it is discarded. After all the comparisons are made with a specific tuple it is removed from the node by an expiry message. Finally, a punctuation mechanism with very little overhead is used, to sort the output data and offer a deterministic result.

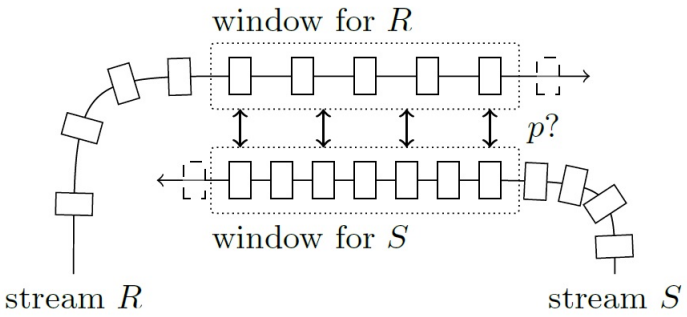


Figure 2.2: *Handshake Join*

SplitJoin [34]

One of the latest implemented algorithms for stream join is SplitJoin algorithm. SplitJoin main contribution is the split of the sequential operation model. More specifically instead of following the classic process, first store and then process the newly incoming tuples, SplitJoin abstract these computation steps as two concurrent and independent steps ("storage" and "processing"). This splitting allows parallel execution and high scalability, which can be further improved by dividing the sliding window into a set of disjoint sub-windows each one assigned to a different join core through a distribution tree. In addition to the splitting of the join computation, SplitJoin algorithm introduces a single top-down data flow instead of the bi-directional data flow models (Handshake Join). This way there is no linear latency overhead as the join cores are no longer chain connected but completely independent, avoiding as well the inter-core communication overhead.

2.3.2 Hardware Based Implementations

Handshake Join on FPGA [45]

The basic idea of Handshake join is to execute window based stream join operation on two streams flowing in opposite directions in order to achieve high scalability and parallelization. To implement this concept on hardware, three issues have to be taken into account. First of all, the result collection, second the capacity of output channel and third the limitation of internal buffer sizes. The core of the proposed architecture is the

join core which is the most fundamental component of the architecture. Its purpose is to evaluate the join condition over the input tuples and generate the output that form the final result. The generated results are then driven into a network of mergers which merge the valid result data streams into one. This network is in fact a binary tree of mergers, whose number is determined by the number of join cores. Finally an admission control is used in order to avoid result losses due to buffer overflows. An overview of handshake implementation in reconfigurable logic is shown in Figure 2.3 below.

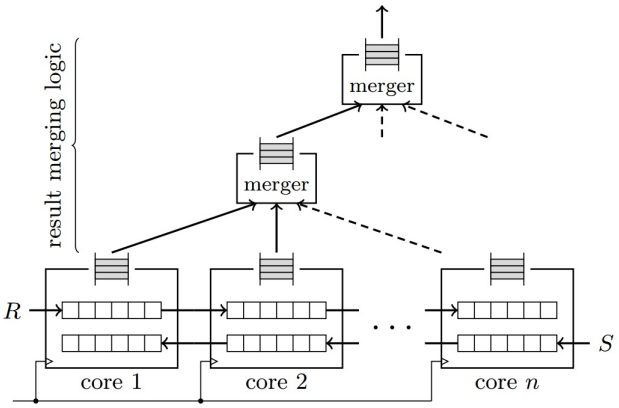


Figure 2.3: *Handshake Join Architecture on FPGA*

Handshake Join Operator on FPGA [35]

An improvement of the Handshake join implementation on FPGA is the Handshake Join Operator. Although it is stated in [45] that the merging network supports high degrees of parallelism and that it is highly scalable, the truth is that it is an overwhelming bottleneck for scalable performance. Due to this fact, an adaptive merging network is proposed, which removes the FIFO buffers in the network and it introduces a ring structure, which is directly connected to the join cores. This way, an output result tuple from a join core is always forwarded and stored in a FIFO buffer (before the merging network). It is important to mention that the chosen buffer FIFO does not need to be at the closest path where the join took place. This way the system avoids buffer overflows. The top-level of this implementation of Handshake join is shown in Figure 2.4.

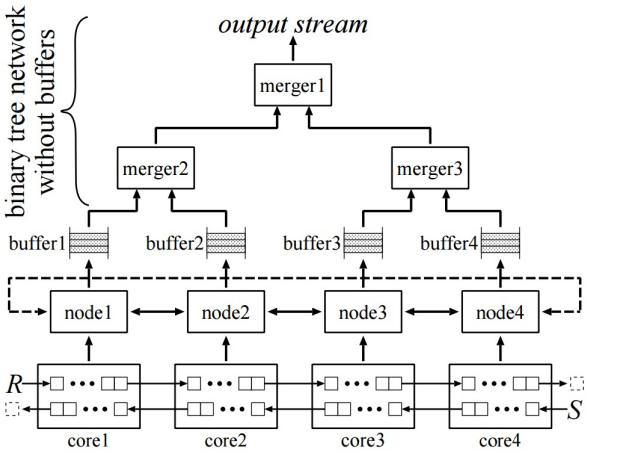


Figure 2.4: *Handshake Join Architecture on FPGA*

ScaleJoin Architecture on FPGA [29]

The first implementation of ScaleJoin algorithm on reconfigurable logic, achieves at least four times higher throughput over the original multi-thread software solution. The architecture consists of two processing elements, which in their turn consist of N processing units connected in a pipeline way. Each processing element is responsible for comparing the newly arrived tuples of one stream with the tuples of the opposite stream inside a window and each processing unit takes care of the comparison. The process is broken in

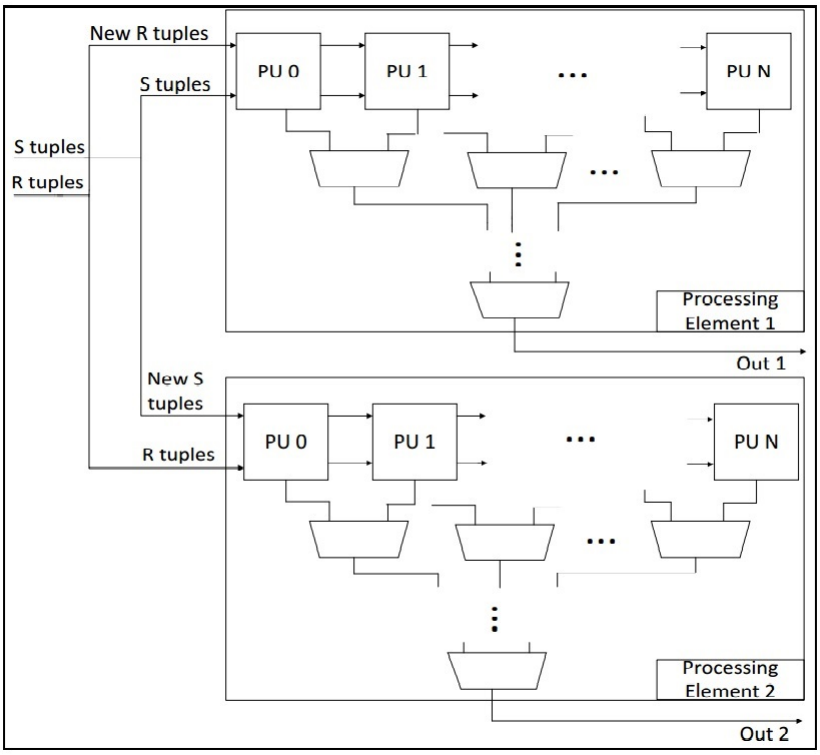


Figure 2.5: *ScaleJoin Architecture on FPGA*

three stages. First, the newly arrived tuples are loaded in the processing units, second the tuples are streamed through the pipeline of processing units and finally, the results are written back in memory. This implementation offers high throughput due to the large number of processing units available (128 in each processing element), as well due to the high parallelization upon the overall comparisons.

Chapter 3

ScaleJoin Algorithm Analysis

In this chapter we are going to analyse ScaleJoin algorithm, one of the most efficient algorithm in data stream join operations. A detailed description of how ScaleJoin algorithm works, its basic components and the performance results are some of the main features that will be presented. Also, the first hardware implementation of ScaleJoin is going to be analysed, in order to note the improvements needed. Finally, based on some observations we propose our architecture, which improves the first hardware-based implementation offering higher throughput.

3.1 The ScaleJoin Algorithm

ScaleJoin algorithm [23], is a state-of-the art stream join algorithm capable of processing efficiently bursty and rate-varying data streams. It offers deterministic and skew-resilient stream processing, while also enabling fine-grain parallelism to achieve high throughput and low latency stream processing. It is based on an abstract data structure, ScaleGate, which operates at the articulation points maintaining the tuples being consumed and produced in a deterministic fashion regardless of the number of processing units or the number of physical streams delivering to them. ScaleJoin allows for the parallel execution of an arbitrary number of sequential stream joins while distributing the overall work among the available processing threads, without assuming any centralized coordinator.

3.1.1 Problems and challenges

The main problems and challenges that ScaleJoin algorithm has to face are deterministic processing, disjoint parallelism and skew-resilience. Below is a brief description of foregoing challenges/problems.

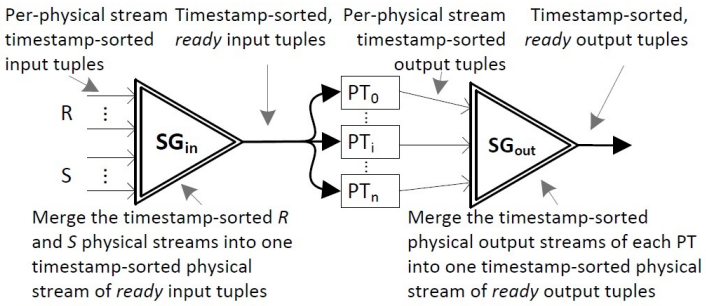


Figure 3.1: *Overview Of ScaleJoin's Architecture*

Deterministic Processing

Given the same inputs in a method, the same outputs must be produced independently of its environment. This requirement allows for a parallel stream join to be leveraged in sensitive scenarios (fraud detection or business-centric pricing applications) and to leverage fault tolerance mechanisms, in which deterministic processing ensures consistency among primary and replica nodes.

Skew Resilience

The ability to process any varying rate of incoming tuples independently by each processing thread. Also, the process of a possible different number of physical streams delivering tuples should not affect the functionality of the algorithm but tuples' comparisons should be assigned to a unique processing thread.

Disjoint Parallelism

ScaleJoin does not rely on a centralized coordinator to assign the workload on processing units sequentially. Every processing thread available (n total processing threads) executes $\frac{1}{n}$ comparisons splitting evenly the overall workload. This way architectures that support high degrees of parallelism are leveraged achieving higher throughput and lower processing latency.

3.1.2 ScaleGate Data Structure

As we mentioned before, ScaleJoin introduces ScaleGate, an abstract data type responsible of maintaining the tuples being consumed and produced in a deterministic fashion, regardless of the number of processing units or the number of physical streams delivering

to them. Inspired by skip lists, a multi level pointer mechanism is designed and adopted to the ScaleGate requirements. This way, fine-grained synchronization is achieved, which in its turn boosts parallelism. ScaleGate is a lock-free implementation ensuring system-wide progress, by guaranteeing at least one of the threads operating on the data structure to make progress independently of the behavior of other threads. It is also linearizable, meaning that every method call should appear to take effect at some point (linearization point) between its invocation and response. Given these attributes, ScaleGate is able to define a total order in the execution, which is consistent with the real-time ordering of operations, maintaining items in a sorted manner.

3.1.3 ScaleJoin Functionality

As already mentioned and shown in figure 3.1, ScaleJoin allows the parallel execution of an arbitrary number n of processing threads (PTs), each consuming, comparing and matching the input tuples delivered by S and R streams. This process is a modified version of the three-step procedure. To be more specific, the first step, which is is responsibility of the input ScaleGate structure (SG_{in}), is to merge the input tuples of each stream into a single timestamp-sorted stream of ready R and S tuples. A ready tuple is defined as ready to be processed, if assuming a tuple t_i^j to be the i-th tuple from a timestamp-sorted stream j, $t_i^j < merge_{ts}$, where $merge_{ts}$ is the minimum among the latest timestamps from each timestamp-sorted stream j. The ready tuples afterwards are sent by the ScaleGate to the PTs in a round robin fashion. The PTs in their turn compare the incoming ready tuple, using a predicate P, with each tuple of the opposite stream inside a window size interval from the tuple. If P holds, an output tuple is created, combining the tuples from the two streams and it is sent to the ScaleGate at the output of the architecture (SG_{out}). SG_{out} then sorts by timestamp the incoming output tuples and sends them as the final system output.

3.2 First Hardware Implementation of ScaleJoin

A first implementation of ScaleJoin algorithm was proposed in reconfigurable logic about a year ago, outperforming the other state of the art software-based multi-core solutions. The reconfigurable architecture was mapped on Convey HC-2ex hybrid computer equipped with an Intel Xeon E5-2640 processor at 2.5 GHz and four Virtex-6 LX760 FPGAs. The implemented system achieves at least four times better throughput vs. the original ScaleJoin algorithm [23]. In this section we present a top-down analysis of the

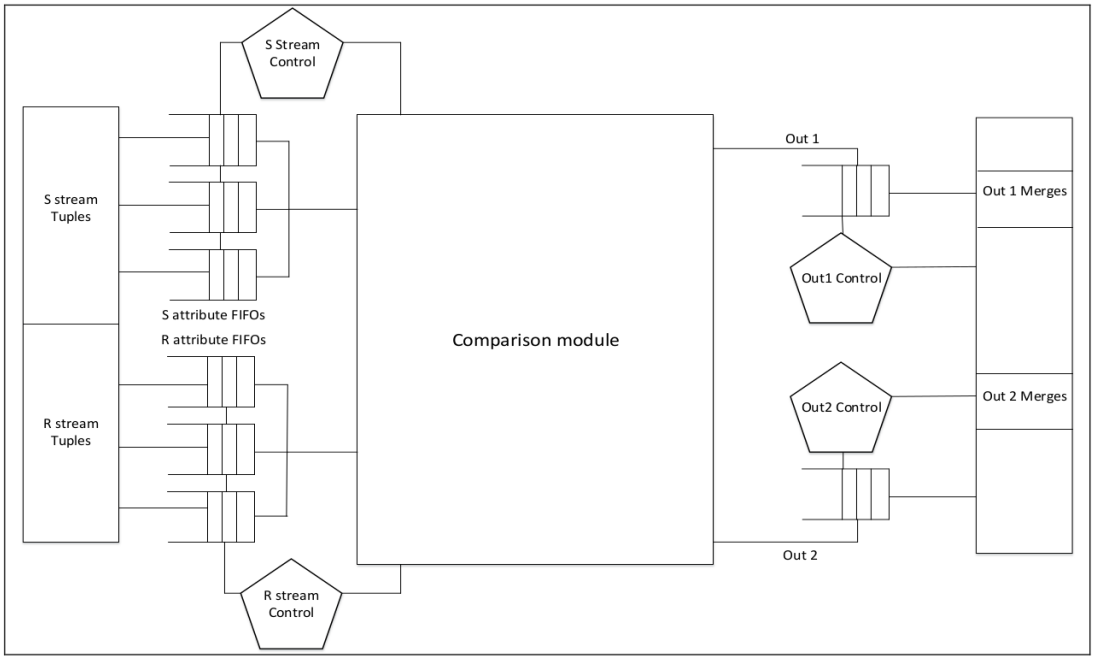


Figure 3.2: *Comparison Module*

first hardware implementation of ScaleJoin algorithm. The top level of this architecture is the Comparison module, shown in Figure 3.2. This unit is mapped inside each of the four FPGAs of Convey HC-2ex hybrid computer. Along with this module, there are several control units responsible for managing the data inputs from the memory controllers of the system. This architecture uses 14 out of the 16 available memory controllers. Only two of the memory controllers utilize the full bandwidth of 64 bits, while the rest utilize only 32 bits.

As we mentioned in Section 2.3.2 Comparison module consists of two processing elements, which in their turn consist of N processing units connected in a pipeline way. In this case, each processing element has 128 PUs, which means that there is a total of 256 PUs in each Comparison module. Each processing element is responsible for comparing the newly arrived tuples of one stream with the tuples of the opposite stream inside a window and each processing unit calculates the corresponding correlations. The process is split in three stages with the first one being the load of the new incoming tuples to the corresponding processing element. After that, the old tuples of each stream are streamed through the pipeline of PUs so that all the comparisons between the tuples take place. If the processed tuples produce a result, this result's information is kept into a FIFO at each PU. Then this information is forwarded through a network of multiplexers to the output

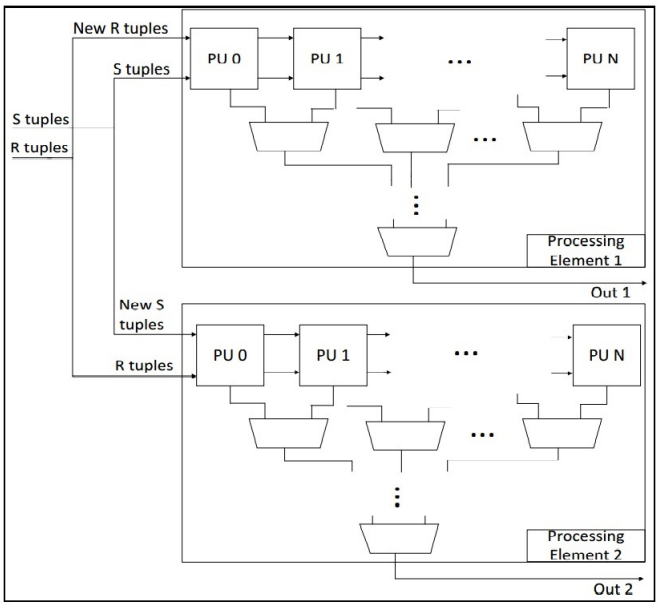


Figure 3.3: *Processing Elements*

in order to be stored back to memory. When all the tuples from the processing window are streamed and all the results are stored to memory the process finishes. This process takes place again every second or many times during a single second, in case the newly arrived tuples are more than the available PUs. This whole process is controlled by several control units that manage the input and streaming of the tuples, as well as the possible stalls and results.

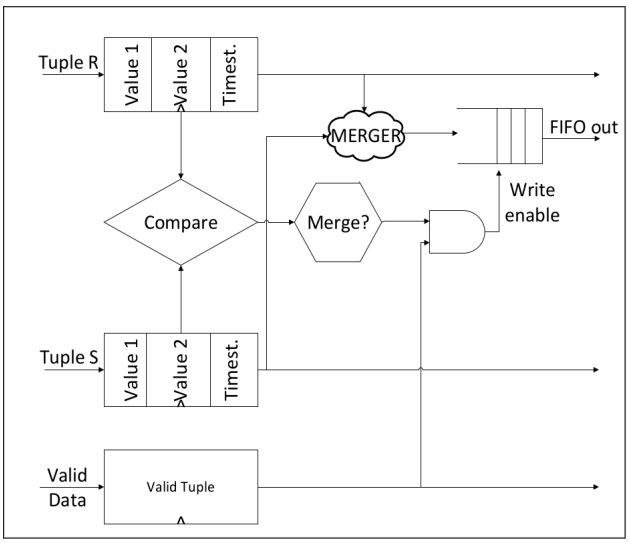


Figure 3.4: *Processing Unit*

Finally, the innermost unit of this architecture is the processing unit (PU). This module is responsible for comparing the tuples of the input streams. If the comparison is successful

a new tuple is created, that hold information for the achieved correlation. This new tuple is kept in a FIFO inside the PU, until the global control forwards it to the output, so that it is stored to memory. Figure 3.4 shows the architecture of the PU.

3.3 Thoughts For An Improved Hardware Implementation

In this section we are going to present the features that limit the performance of the first hardware implementation of ScaleJoin algorithm and propose our thoughts for a new design that will improve the existing performance.

3.3.1 Improvements Needed

Although the first architecture achieves high throughput and outperforms the state of the art software-based implementations, further improvements can be made to accomplish even greater performance results. The main reasons that limit the performance results are:

- The clock frequency achieved, as well as the timing errors of the first hardware implementation confine the performance. The reconfigurable part of Convey's HC-2ex can reach a clock frequency of up to 150 MHz, something that would significantly leverage processing throughput.
- The resources utilization of the first implementation, as shown in Table 3.1, is less than 30% of the available resources Convey HC-2ex offer. This was due to routing issues of the design, that could not let a larger design to be mapped.
- Only 14 out of the 16 available memory controllers offered by the platform are used and only two of them use the full bandwidth. A better management of the memory controllers has to be made, in order to take advantage of the full bandwidth of all the available memory controllers.

FPGA Available Resources	System Utilization
DSPs	0/864 (0%)
Slice Registers	158777/948480 (16%)
Slice LUTs	151688/474240 (31%)
Block RAMs	114/720 (15%)

Table 3.1: *System Resources Utilization*

In order to address the above mentioned issues, a new architecture should be proposed that will overcome the problems that limit the first hardware implementation of the

ScaleJoin algorithm.

3.3.2 Design of Proposed Architecture

Based on the observations we made on the first hardware implementation of ScaleJoin algorithm, as well as on the software algorithm itself, we analysed the way our implementation is structured, its processing steps and how we plan to overcome the issues that appeared in the first hardware implementation.

Clock Frequency

Regarding the clock frequency we had to locate the longest data path delay (critical path), which is responsible for the system's clock frequency and the timing errors that appeared. We concluded that the critical path was inside the implemented processing unit (PU) due to the comparison that takes place there. To address this issue we chose to alter PU's implementation and build a pipelined architecture, in order to reduce the critical path and increase the clock frequency.

Resources Utilization

As we have already mentioned, the low resources utilization of the first hardware implementation was due to routing issues that restricted the mapping of a larger design. To address this issue, we propose a different architecture based on the first hardware implementation. To be more specific, we divide the pipeline architecture of the Processing Element into a parallel pipelined architecture. This way, we increase the level of parallelism, we split the process workload and use smaller networks of multiplexers to forward the results while being able to map more PUs and a larger design. The algorithm's steps are kept as described in Section 2.3.2. Based on these steps an abstract representation of the process that we follow is presented on the flowchart in Figure 3.5.

Memory Controllers Management

Finally, regarding the memory controllers a better management has been made that exploits all 16 memory channels of 64 bit bandwidth. In the first hardware implementation not all the memory controllers were used and most of them used only half of the available bandwidth (32 bits). So the concept in our implementation is every channel to use all of the 64 bits available and to make a better arrangement of the channels to exploit their full potential.

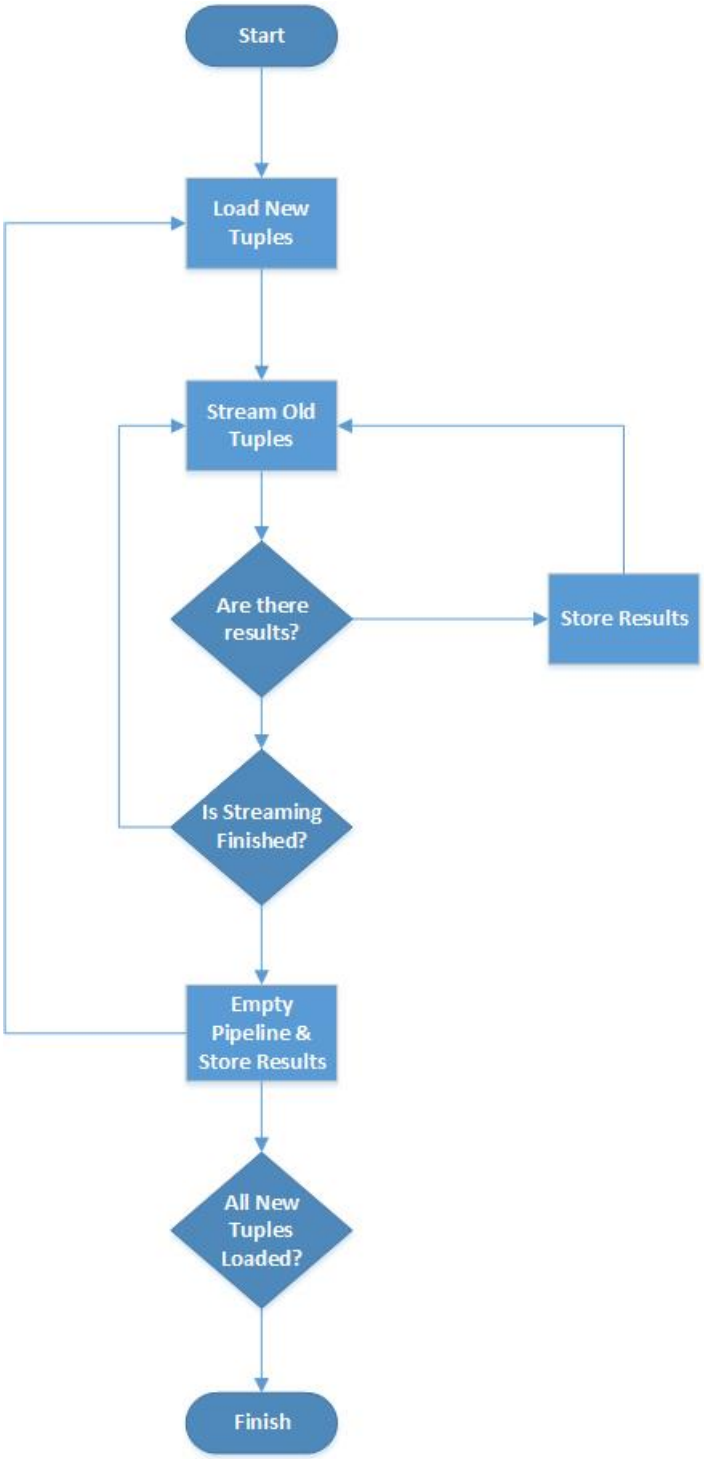


Figure 3.5: *Process Flowchart*

Chapter 4

ScaleJoin System

This chapter presents the architecture of the ScaleJoin algorithm in reconfigurable logic. The architecture is mapped on Convey HC-2ex hybrid computer equipped with two six-core Intel Xeon E5-2640 processors running at 2.5 GHz and four Virtex 6 LX760 FPGAs. This chapter is organized as follows: reference to Convey HC-2 platform and its tools in Section 4.1. The top-level and the way data is imported into the system is explained in Section 4.2. Section 4.3 presents a top-down analysis of the ScaleJoin module and all of its sub-components is presented. Finally, the way the results are stored back to memory is explained in Section 4.4.

4.1 Convey HC-2 Platform

Convey Computers made a revolution to the high-performance computing (HPC) field by launching the world's first hybrid-core computer HC-2 that breaks the barriers of expensive power, performance and programmability. The HC-2 server managed to change the till today known HPC as it breaks through the current power/performance wall to significantly increase performance for certain compute and memory bandwidth intensive applications. Also, it is easy for programmers to use as it provides full support of an ANSI standard C, C++ and Fortran development environment and it significantly reduces support, power and facility costs for companies. Convey with HC-2 managed to fill a market space called hybrid-core computing, which marries low cost and simple programming model of a commodity system with the performance of customized hardware architecture.

4.1.1 System Architecture

Convey Hybrid Core systems utilize a commodity motherboard that includes an Intel 64 host processor and standard Intel I/O chipset, along with a reconfigurable coprocessor

based on FPGA technology. This coprocessor can be reloaded dynamically while the system is running with instructions that are optimized for different workloads. It includes its own high bandwidth memory subsystem that is incorporated into the Intel global memory space, creating the Hybrid-Core Globally Shared Memory (HCGSM). Coprocessor instructions can be thought of as extensions to the Intel instruction set – an executable can contain both Intel and coprocessor instructions and those instructions exist in the same virtual and physical address space.

The coprocessor supports multiple instruction sets (referred to as "personalities"). Each personality includes a base set of instructions that are common to all personalities. This base set includes instructions that perform scalar operations on integer and floating point data, address computations, conditionals and branches, as well as miscellaneous control and status operations. A personality also, includes a set of extended instructions that are designed for a particular workload. The extended instructions for a personality designed for signal processing, for instance, may implement a SIMD model and include vector instructions for 32-bit complex arithmetic.

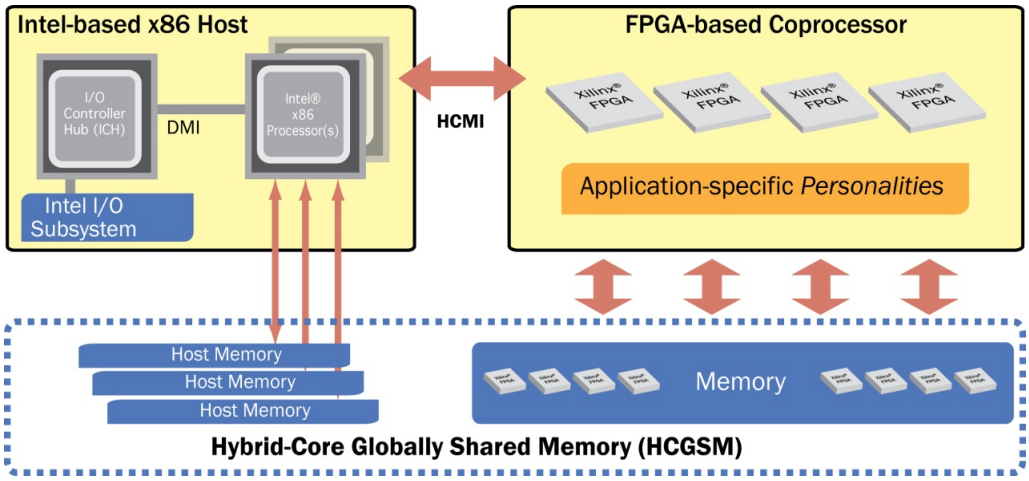


Figure 4.1: *Convey Hybrid Core System Diagram*

4.1.2 Coprocessor

The coprocessor has three major sets of components, referred to as the Application Engine Hub (AEH), the Memory Controllers (MCs), and the Application Engines (AEs). The AEH is the central hub for the coprocessor. It implements the interface to the host processor and to the Intel I/O chipset, fetches and decodes instructions, and executes scalar instructions (see Personalities below). It processes coherence and data requests from the

host processor, routing requests for addresses in coprocessor memory to the MCs. Scalar instructions are executed in the AEH, while extended instructions are passed to the AEs for execution.

To support the bandwidth demands of the coprocessor, 8 Memory Controllers support a total of 16 DDR2 memory channels, providing an aggregate of up to 80GB/sec of bandwidth to ECC protected memory. The MCs translate virtual to physical addresses on behalf of the AEs, and include snoop filters to minimize snoop traffic to the host processor. The Memory Controllers support standard DIMMs as well as Convey designed Scatter-Gather DIMMs. The Scatter-Gather DIMMs, are optimized for transfers of 8-byte bursts, and provide near peak bandwidth for non-sequential 8-byte accesses. The coprocessor therefore not only has a much higher peak bandwidth than is available to commodity processors, but also delivers a much higher percentage of that peak for non-sequential accesses.

Together the AEH and the MC's implement features that are present in all personalities. This ensures that important features such as memory protection, access to coprocessor memory, and communication with the host processor are always available.

The Application Engines (AEs) are four user-programmable Virtex-6 XC6VLX760 FPGAs, which are the heart of the coprocessor and implement the extended instructions that deliver performance for a "personality", which is a particular configuration of these FPGAs. The AEs are connected to the AEH by a command bus that transfers opcodes and scalar operands, and to the memory controllers via a network of point-to-point links that provide very high sustained bandwidth. Each AE instruction is passed to all four AEs. How they process the instructions depends on the personality. For instance, a personality that implements a vector model might implement multiple arithmetic pipelines in each AE, and divide the elements of a vector across all the pipelines to be processed in parallel.

4.1.3 Personalities

A personality defines the set of instructions supported by the coprocessor and their behavior. A system may have multiple personalities installed and can switch between them dynamically to execute different types of code, but only one personality is active on a coprocessor at any one time. Each installed personality includes the loadable bit files

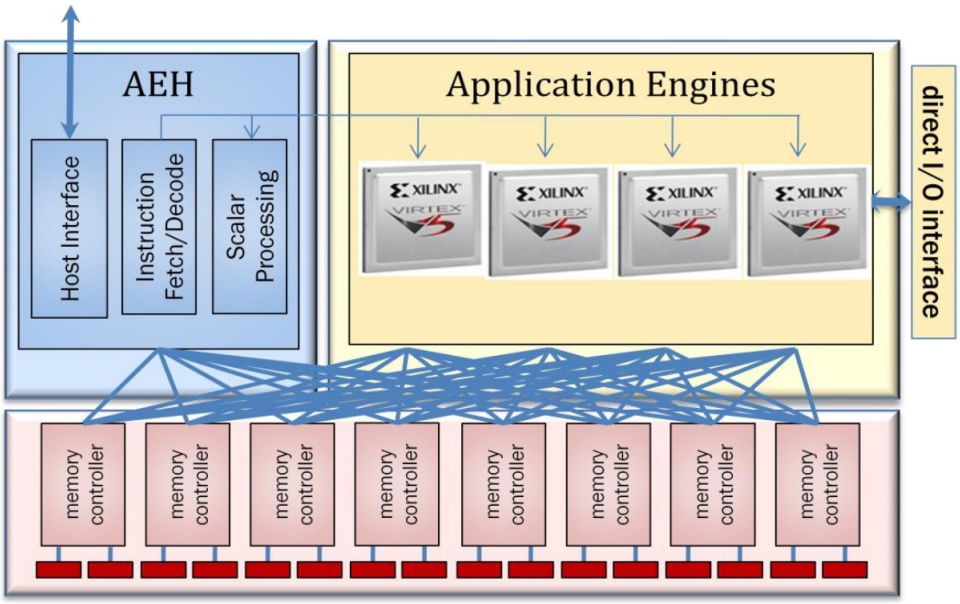


Figure 4.2: *Coprocessor Diagram*

that implement a coprocessor instruction set, a list of the instructions supported by that personality, and an ID used by the application to load the correct image at runtime. All personalities implement a base set of instructions referred to as the scalar instruction set. These instructions include scalar arithmetic, conditionals, branches, and other operations required to implement loops and manage the operation of the coprocessor. In addition to the scalar instructions, each personality includes extended instructions that may be unique to that personality. Extended instructions are designed for particular workloads, and may include only the operations that represent the largest portion of the execution time for an application. For instance, a personality designed for seismic processing may implement 32-bit complex vector arithmetic instructions.

All personalities have some elements in common:

- Coprocessor execution is initiated and controlled via instructions, as defined by the Convey Instruction Set Architecture.
- All personalities use a common host interface to dispatch coprocessor instructions and return status. This interface uses the globally shared memory.
- Coprocessor instructions use virtual addresses compatible with the Intel® 64 specification, and coherently share memory with the host processor. The host processor and I/O system can access coprocessor memory and the coprocessor can access host memory. The virtual memory implementation provides protection for process address

spaces as in a conventional system.

- All personalities support a common set of instructions called the scalar instruction set. These instructions implement basic scalar and control flow operations required to implement interfaces and manage the operation of the AEs.

These common elements ensure that compilers and other tools can be leveraged across multiple personalities, while still allowing customization for different workloads.

4.2 Reconfigurable ScaleJoin System

In this section, we analyse the top level architecture of our system and the way the data is imported. We explain how memory controllers are configured to read and write in Convey's shared memory and how C and Assembly code help our system to work consistently. Our system uses all four FPGAs available of the Convey HC-2ex supercomputer as also all of the 16 memory controller channels. Figure 4.3 presents the total top level of our system architecture for the stream join processing including the shared memory and the 4 FPGAs.

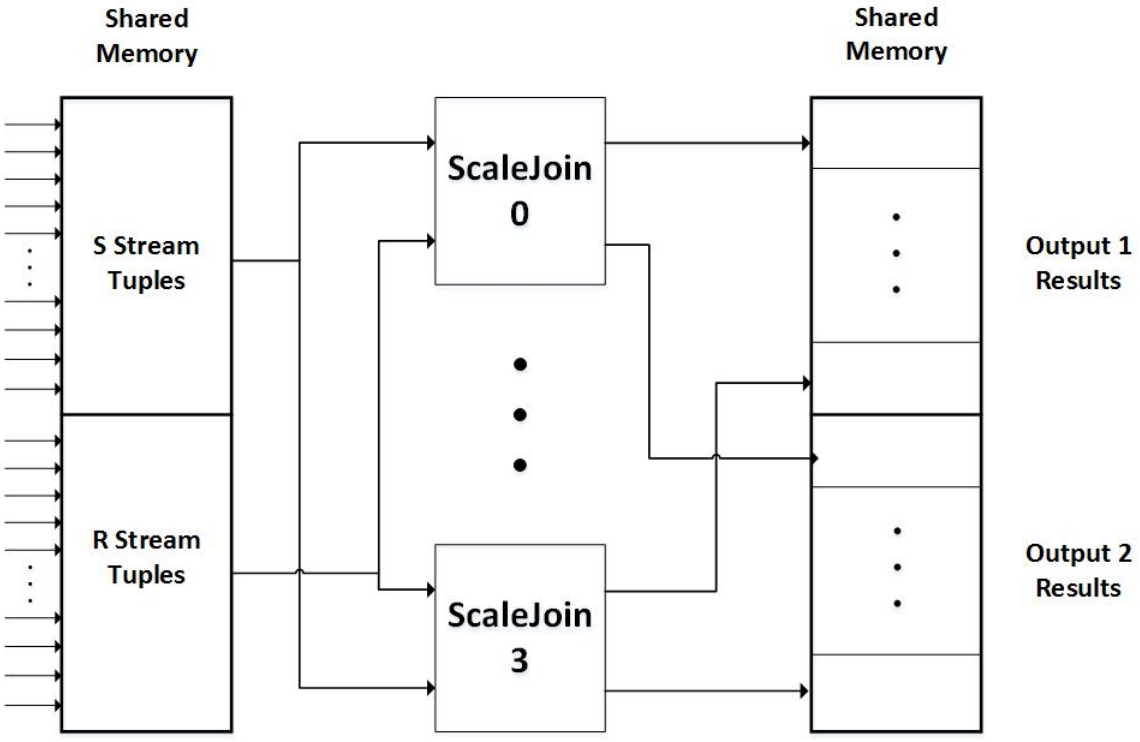


Figure 4.3: *System Architecture Top Level*

4.2.1 C Code

In the first place, we refer to the C code and its role to our system. Inside C code we generate random 96 bits width tuples for both streams and we store them in Convey's RAM. These tuples afterwards, are copied in Convey's shared memory so that we can access them through the memory controllers (see section 4.2.2). At this point we have to mention that because the tuples are 96 bits wide, we split them into 64 bits in order to be able to store them in Convey's shared memory. It has to be noted, that there is no data loss, as the remaining 32 bits from one tuple, are concatenated with the 32 LSB of the next tuple. This way, we have three values stored in shared memory which represent two tuples. To retrieve the tuples from the stored 64 bits values we follow the reverse process, as we will explaine below in Section 4.3.1.

Furthermore, the C code needs to determine the window size and the rate of the new tuples that come up every second. The window size and the rate of new tuples, remain the same for both streams throughout the process and determine the computational cost. To be more specific, assuming we have a window size $W = 10$ *minutes* and a rate $R = 500$ *tuples/second*, since each tuple from the one stream has to be compared with all the tuples of the other stream inside W, the total number of comparisons per time unit is: $2 \times W \times R^2 = 300$ *million comparisons/second*.

Moreover, the C code is responsible for initializing our system using a *copcall* routine. This routine triggers the system while introducing as arguments the necessary inputs that our system needs to run successfully. We use a total of 10 arguments including the address for each stream, the window size, the result address, the number of tuples to be processed and others. *Copcall* routine is called iteratively until all data have been processed.

4.2.2 Assembly Code

Along with the C code, there is an Assembly code, which serves as the link between the software and the hardware. As we explained in the Section 4.2.1, a *copcall* routine is called every time the systems needs to run. The arguments of this routine are processed by the Assembly code and they are driven in our hardware top-level. This process is done by storing all the arguments needed by our system to registers accessible from the hardware.

Due to the fact that Convey HC-2ex has 16 memory controller channels, the Assembly code is responsible for computing the exact address each channel has to read from (or write to). Since the number of the tuples to be processed is known every time the system runs,

we can easily calculate the address limits of each memory controller. Also, every argument containing a value (and not an address), is stored in the same way into registers which serve as inputs to our system. Finally the Assembly code "splits" the process into the four FPGAs of Convey HC-2ex supercomputer by dividing the data to be read in each FPGA respectively. By using all 16 memory controllers' channels and the four FPGAs, we exploit the high level of parallelism hardware can offer.

4.2.3 Memory Controllers

Convey HC-2ex supports 16 memory controller channels, which allow access to physical memory by 8-byte blocks instead of 64-byte cache lines (as the host does). The memory access using 8-Byte blocks reduces the inefficiencies encountered when accessing memory by no unity strides (or randomly) with a cache-based system. The inefficiency can be as drastic as 1/8 of the peak bandwidth, because if only 4 or 8 bytes out of an entire 64-byte cache line are needed, the rest of the transfer is wasted. In this section we will describe how we configure these memory controllers to retrieve the data needed to feed our system.

To begin with, each memory controller has its own FSM, which serves as a control unit of the memory controller. Thus, we have 16 FSMs responsible to assign the correct address to every channel, either this channel has to read or write to the shared memory. We have already mentioned the purpose of our algorithm is to compare the new tuples of one stream with all the tuples of another stream within a window size and vice versa. To accomplish this, we use the eight channels for the comparisons of one stream with all the elements of the second stream and the other eight channels take over the comparisons, which are related to the other stream. For our implementation we used two channels to read the new tuples (one for each stream) and the rest 14 to read the old tuples (seven for each stream). The channels responsible for reading the new tuples are also the ones which write the results to the shared memory. This happens because after loading the new tuples, these channels become inactive allowing us to use them for storing the results to memory.

4.3 ScaleJoin Implementation

This section presents a top-down analysis of our implementation of ScaleJoin algorithm in reconfigurable logic. In order to make this algorithm work correctly, we implemented our design in Convey HC-2ex supercomputer. This way, we take advantage of the high scalability and performance advantages hardware can offer, such as fast I/O data links

and a large amount of hardware resources. The ScaleJoin module is the top-level of our implementation and it is mapped inside each of the four FPGAs. It consists of two modules (see 4.3.2), each one responsible to make the necessary comparisons as we have already explained. Figure 4.4 presents the ScaleJoin module.

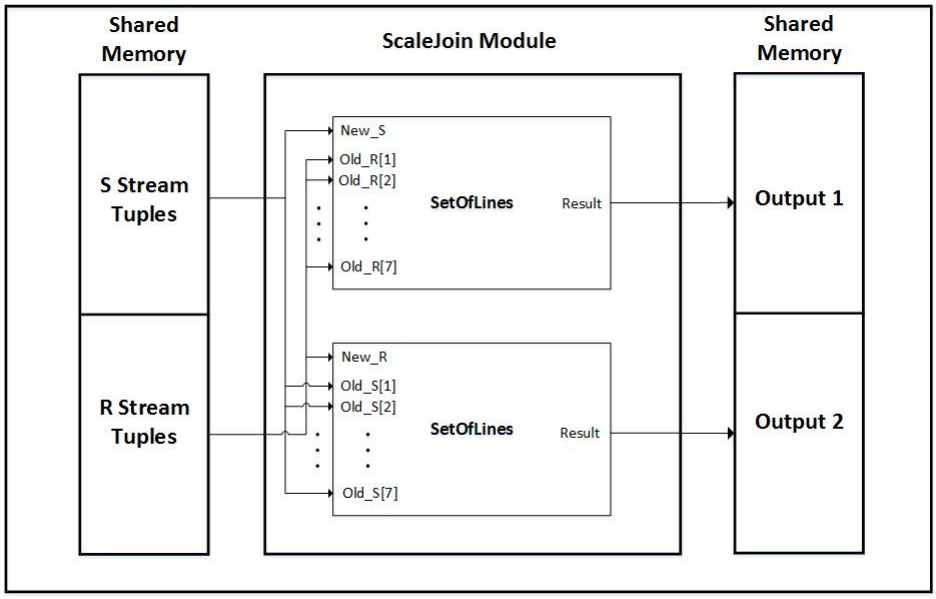


Figure 4.4: *ScaleJoin Module*

To begin with, a C code is used to read input tuples for both streams and store them in Convey's RAM, as described in the previous section. Also, the C code is responsible to feed our system with these tuples given a Rate size R and a Window size W, which we also mentioned in section 4.2.1. Rate and window sizes, as also the number of tuples to be processed each time, are essential for our system to run correctly, as they determine the size and execution time of the process.

The whole system process is divided in steps. The first step loads the new tuples into our system. When all new tuples have been loaded, the old tuples are streamed until all comparisons are carried out. In the third step, the results that may have occurred are written back in the shared memory. This process is followed every time the system runs and it is shown in figure 4.5.

4.3.1 Data Inputs

As we previously mentioned, the memory controllers that Convey provides, have a maximum bandwidth of 64 bits. In our implementation, we use the whole width of the memory controllers to fetch the data from the shared memory, as also to store back the results. At this point, we have to indicate a problem that we had to solve. The width

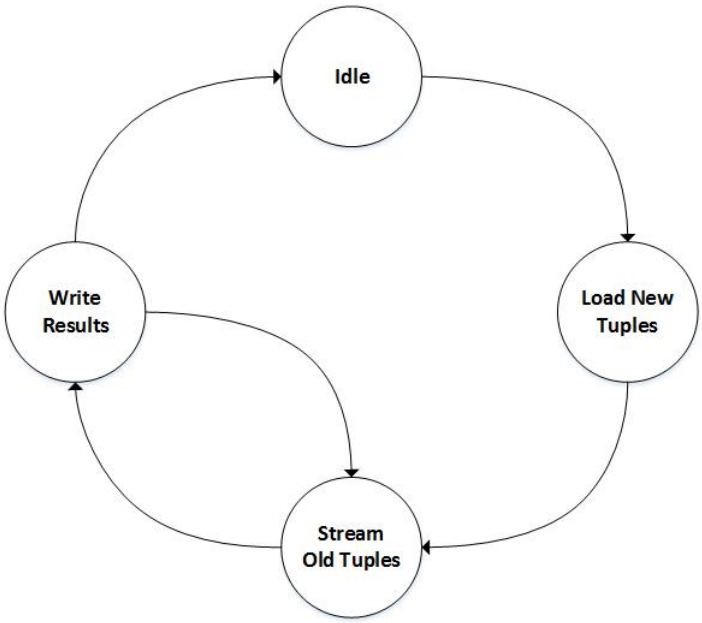


Figure 4.5: *ScaleJoin Control*

of the tuple is 96 bits, which is 32 bits bigger than the size of the data lines, which are used by the memory controller. To overcome this issue, we implemented an FSM, which is responsible for merging the incoming 64 bits data from the memory controller into 96 bits tuples. Our FSM is used after each memory controller channel and has three states. In the first state, we store the incoming 64 bits data into a register. In the second state, we merge the 32 LSB of the following incoming data from the memory controller with the previously stored 64 bits from the first state so as to have a complete tuple. We also keep the 32 MSB to a different register. In the third state, we merge the 64 bit upcoming data with the remaining 32 bits, which were stored in the previous state. Every time a tuple is created, it is driven into a FIFO where it is stored. This way in three states, i.e three clock cycles, we can have 2 complete tuples. Along with the tuples, a valid bit is produced, which serves as a write enable to the FIFOs where the tuples are stored.

4.3.2 SetOfLines Module

In this section, we refer to the SetOfLines module. There are two SetOfLines modules inside each FPGA chip The SetOfLines module is responsible for comparing the new tuples of one stream with all the tuples of the other input stream in a given window size. In our implementation, the SetOfLines module consists of seven lines of process, which consist of N processing units (PUs). The number of PUs in each line is configurable, which means that we can change it depending on the resources available in our system. The number of lines is based on the the fact that we have 8 memory controller channels for each

SetOfLines module. As we also mentioned in Section 4.2.3, one memory controller channel is responsible for reading the new tuples and storing the results to the shared memory and the other seven for reading the old tuples. We chose to assign more channels for the old tuples as they comprise the biggest percentage of the computational cost. This way we achieve higher level of parallelism and the process load is distributed to each line, allowing us to make the necessary comparisons faster.

The tuples are read and stored in FIFOs, as explained in Section 4.3.1. There is a total of nine FIFOs in each SetOfLines module as one holds the new tuples, seven hold the old tuples and one holds the results. The FIFO that holds the new tuples, sends its contents to all seven lines in contrast with the seven FIFOs that hold the old tuples, which drive their contents into each line, respectively. This way each line is loaded with the same new tuples, whereas the old tuples are split and streamed in each line, making the processing 7 times faster.

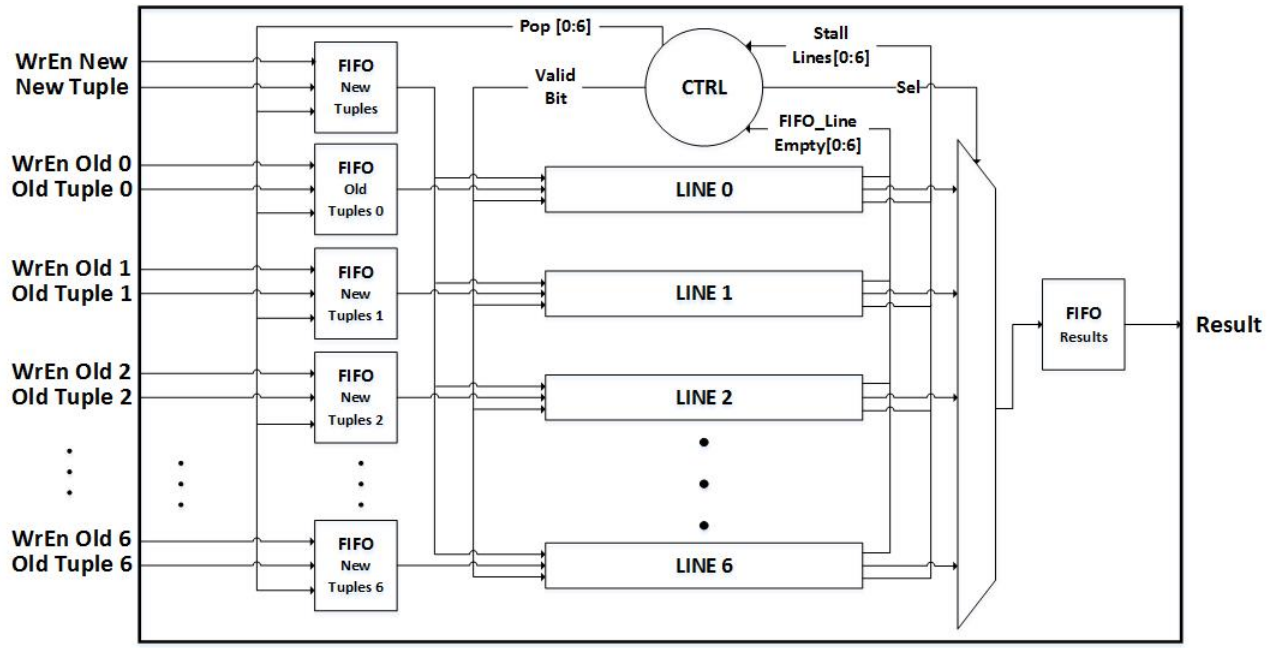


Figure 4.6: *SetOfLines Module*

Furthermore, we had to implement nine control units (which are actually FSMs). One control is used to manage the new tuples, seven are used for the old tuples of each line and one last control for sending out the results. To be more specific, the first step when the system starts to execute, is to load the new tuples to each line. This is the responsibility of the first control unit which also checks when all the new tuples are loaded or if the available PUs in the lines are occupied. If any of these happens the execution passes to next state, which is the streaming of the old tuples. At this point, the seven control units

for the old tuples take over by feeding each line with data. Along with these seven FSMs, FSM, which runs simultaneously and it controls the results that come out. This control searches each clock cycle in a round robin fashion, if there is a result available in any of the lines. If there is, the result is stored in the Results FIFO in order to be stored afterwards in the shared memory. It is also important to mention that when the execution has ended and all the comparisons have been made this FSM continues to run until all the results from the lines have been stored to the Results FIFO. After this is done the control unit sends a *finish* signal in order to bring the next new tuples and continue processing. A representation of how SetOfLines module is structured is shown in Figure 4.6.

4.3.3 LineOfPUs Module

Moving down on our architecture's hierarchy, we refer to the LineOfPUs module. It consists of N processing units(PUs), where the comparisons take place, a control unit, which is responsible for the correct function of the module, a network of multiplexers and a FIFO for storing the results. The LineOfPUs module with all of its components appears in Figure 4.7.

4.3.3.1 Parallelization Of Processing Units

Stream join processing algorithms have a need for high processing throughput, which is constantly increasing. In order to achieve high throughput, the ScaleJoin algorithm uses many processing units, via the parallelization of the whole processing on tuples given and therefore parallelization upon the overall comparisons. So, in order to decide which parallelization technique we are going to use to our processing units, we need to think about satisfying the parallelization challenges of the ScaleJoin algorithm. Furthermore, we need to have exactly the same outcome as the official algorithm for the same input datasets.

Therefore, out of many parallelization techniques, we decided to use a pipeline implementation, with a pipe length equal to the maximum incoming tuple rate per second of the whole process. Figure 4.7 shows a set of processing units, which is a number of processing units that are connected to each other in a pipeline fashion. To increase the parallelization level even more, we used seven lines of process as showed in Figure 4.6.

As it comes to the functionality of the module, in each cycle we load each processing unit with a new tuple from the one stream. When we finish loading the new tuples, we feed each processing unit with the older tuples from the opposite stream inside a Window size

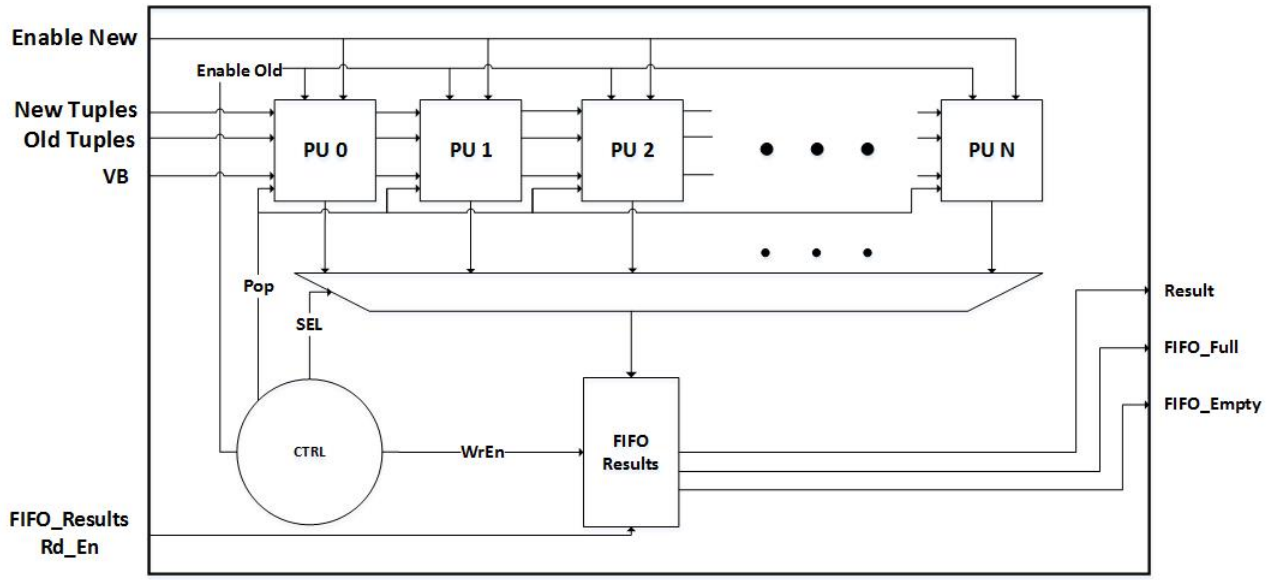


Figure 4.7: *LineOfPUs Module*

W, in order to make all the appropriate comparisons that need to be done in a processing second. The opposite stream's tuples are stored in one processing unit, they are processed and then they move to the next processing unit in a pipeline fashion.This process repeats with a view to compare all the old tuples with all the new ones (sliding window). So, at the end of each cycle, all the possible comparisons have been made and the results are stored in the respective processing unit's FIFO. Thus, all the possible correlations among the newly arrived tuples and the previously arrived tuples are computed. Finally, each result passes through a network of multiplexers and is stored in the FIFO that keeps the results. When all tuples from the processing window are streamed and no other results have to be stored, the processing stage finishes. The above process takes place again at each second or many times during a single second in case the newly arrived tuples are more than the available PUs of the LineOfPUs.

4.3.3.2 LineOfPUs Control Unit

The control unit that lies inside the LineOfPUs is maybe the most important unit not only of LineOfPUs module but of our whole system. This unit manages the execution process by checking for stalls, results that may have arrived and termination conditions. An abstract presentation of what this control is responsible for is available in Figure 4.8.

As we can easily understand, this module is an FSM, which has as inputs the output signals of its controlling components. Being in an idle state, our FSM waits for a *Start* signal to start streaming the old tuples, when all the new tuples have been loaded in the

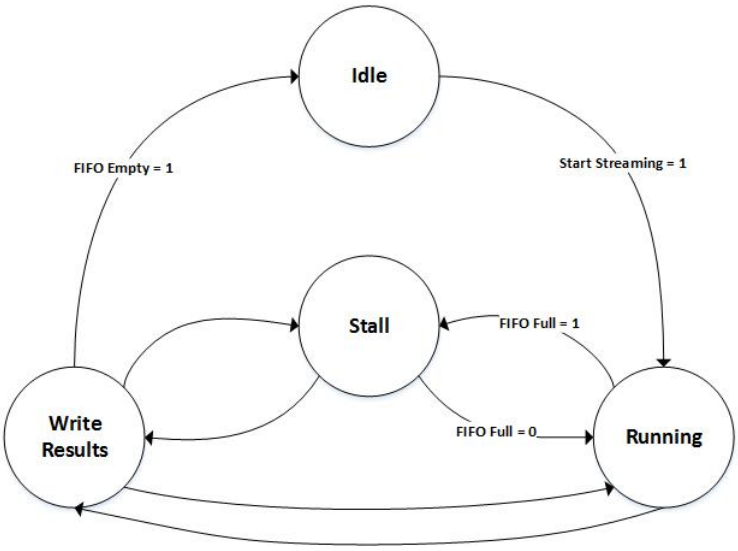


Figure 4.8: *LineOfPUs Control Unit*

available PUs. Then, the FSM requests for old tuples to feed the pipeline in every clock cycle, until all the comparisons have been made or a stall is requested. An important task, is the storage of the results that appear inside the FIFO of each PU. This is done by searching and reading each processing unit's output in a round robin fashion with a way not to hold our implementation stalled for too long. If any of the PUs or the FIFO that keeps the results is filled, a *FULL* signal is sent to our control to stall the process. This way, we ensure that no result is lost during our system's execution. During the stall state, our system continues to check every PU for results and if one is found, it will be driven through the network of multiplexers in the FIFO that keeps the results. We keep staying in Stall state until no FULL signals come as input to our control. We have to mention here that if the FIFO that keeps the results becomes FULL, we can not search the PUs for results as they have no space to be stored and they will be lost. In this case we wait for the control to manage the results of each line and empty space in our FIFO. Finally, when all the comparisons take place our control keeps checking all the processing units until they are all empty and all the results are written in the results' FIFO. After that, it returns to Idle state, where it waits for the next execution.

4.3.4 Processing Unit

As we mentioned in previous chapters, the processing units are the fundamental part of the ScaleJoin algorithm. The processing units need to compare two input tuples (one for each stream). If we have successful comparison, the system will create a new tuple by merging all the attributes of both input tuples. This module is fed with tuples from stream

R and stream S, it compares their attributes and it defines if a comparison is successful. Then, it proceeds to the merging phase of the tuples.

In order to create a module that has the same functionality as a processing unit, we first had to decide how we should parallelize the Processing units. The level of parallelization was presented in section 4.3.3.1.

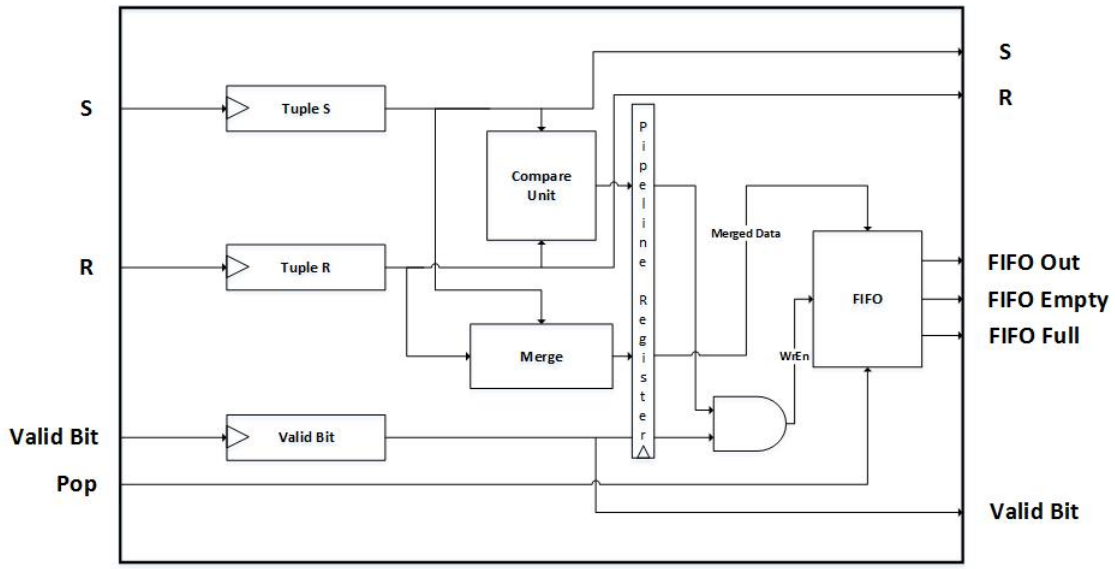


Figure 4.9: *Processing Unit*

In our implementation the processing unit is the core of our system, where all the comparisons of one tuple are made. It consists of two 96 bits registers, where we keep the necessary attributes of each stream's tuple. To be more specific, tuple R is composed of attributes $\langle t_s, x, y, z \rangle$, where x,y,z are of types int, float and char[20] respectively while tuple S is composed of attributes $\langle t_s, a, b, c, d \rangle$, where a,b,c,d, are of types int, float, double and boolean. In order a comparison to be successful the below condition must be valid.

$$|a - x| <= 10 \texttt{ AND } |b - y| <= 10$$

As we can understand from the above expression we do not need all the tuple's attributes in order to make the comparisons and that is the reason we use 96 bits registers. Each 96 bits register keeps the two attributes needed by the comparison i.e (a,b) for tuple S and (x,y) for tuple R as well as the unique timestamps of each tuple. There is also a one-bit register that holds a valid bit. This bit is used to identify, if the comparison held at the specific clock cycle is valid and the result needs to be stored. Furthermore, inside the PU module there is the Compare unit, which is responsible for the correlation-comparison between tuples as, explained previously. We have to mention here that if a successful comparison

exists, then we merge the IDs/Timestamps of these tuples into a new 64 bits tuple, which we keep in a FIFO. This way we know which exact tuples made a successful comparison in order to merge them into a new tuple containing all their attributes $\langle t_s, x, y, z, a, b, c, d \rangle$. Last but not least, in order to achieve high clock frequency we made the architecture of the PU pipelined. To be more specific we split the process in two pipeline stages. The first consists of the registers keeping the tuples and the valid bit as well as the compare unit. The latter includes the FIFO, where the results are kept. A full representation of the processing unit is shown in the Figure 4.9.

Chapter 5

Evaluation

In this chapter, the performance results are presented, analysed and compared with previous works. Section 5.1 describes how we confirmed that the determinism is respected and that the same output is produced given the same input. Section 5.2 analyses the theoretical background and experimental setup and Section 5.3 presents the evaluation of the official software-based algorithm. Finally in section 5.4 we present the performance evaluation and compare our proposed architecture with the other state-of-the-art implementations.

5.1 Results Check

The first step during the evaluation was to make sure that determinism is respected, meaning that given the same inputs the same output results must be produced. In order to do this, we used the same data set as the software-based implementation of ScaleJoin algorithm. This dataset includes the tuples for each stream and the merge results as well as the tuple rate and window size used. The procedure that we followed was to import the input data (tuples for each stream, tuple rate and window size) using the C code mentioned in 4.2.1 and then compare the produced results with those of the original algorithm. When the proper functionality of our implementation was confirmed we were able to test our system to its limits and evaluate its performance.

5.2 Experimental Setup

As we have already mentioned in the previous chapters, the stream join processing purpose is to compare the newly arrived tuples from an input stream with all the tuples of an opposite stream inside a time window. This process takes place for each one of all the

input streams. Considering that the tuples from both streams arrive at a rate R tuples/sec and the processing window has size W the number of total number of comparisons need to be made during a second is:

$$TotalNumberofComparisons = 2 \times W \times R^2$$

Given a window size of standard values (300, 600 and 900 seconds), the above formula helped us determine the maximum rate R (tuples/sec) that our system can process during a single second.

Next, we needed to choose a benchmark to evaluate our new proposed hardware-based system of the ScaleJoin algorithm. Thus, we used the same benchmark that is used by previous works, like the HandShake Join, the CellJoin and the original ScaleJoin algorithm. To be more specific, we used a C code to generate tuples for both streams S and R. In more details, stream S is composed by attributes $\langle t_s, a, b, c, d \rangle$, where a, b, c, d, are of types int, float, double and boolean respectively, while stream R is composed by attributes $\langle t_s, x, y, z \rangle$, where x, y, z are of types int, float and char[20] respectively. The values are randomly generated in a uniform distribution in the interval $1 - 10000$. The predicate used to confirm a valid comparison is the following:

$$|x - a| < 10 \quad AND \quad |y - b| < 10$$

After all the tuples of each second are read, they are copied into Convey's shared memory and loaded in our system (reconfigurable logic part) at every system call. After the process is completed the results are read and presented to the user by the processor. This procedure takes places every second.

In a real-life application, the arrival of data can take place in parallel from parallel running threads, which store the streaming values in different places of RAM. The implemented system can operate with multiple physical links, like the ScaleJoin algorithm. The performance results, which are presented in the next section, are based on a single thread data generation, which is considered to be the worst case in terms of performance.

5.3 Software Evaluation

ScaleJoin was compared with the state of the art stream join algorithms, CellJoin and Handshake Join. The main attributes evaluated were (i) ScaleJoin's scalability in terms

of comparisons/sec (c/s) and tuples/sec (t/s), (ii) ScaleJoin's rate at which tuples can be added to and retrieved from ScaleGate instance and finally (iii) the end-to-end processing latency. The evaluation takes place in two different systems. The first one (S_1) consisted of a 2.6 GHz AMD Opteron 6230 (48 cores over 4 sockets) with a non-uniform memory access (NUMA) and 64 GB of memory while the second system (S_2) was equipped with a 2.0 GHz Intel Xeon E5-2650 (16 cores over 2 sockets) and 64 GB of memory. The data sets, which were used for evaluating ScaleJoin algorithm, were described in Section 5.2. It is important to mention that we used the same datasets for comparing the ScaleJoin implementations with Handshake join and CellJoin algorithms.

First, the throughput rates for the two above systems when they implement the ScaleJoin algorithm were evaluated. ScaleJoin algorithm achieved approximately 4 billion *comparisons/sec* when executed in System 1 and 1.4 billion *comparisons/sec* with a potential to reach 1.9 billion *comparisons/sec* (with balanced workload) when executed in System 2. The throughput regarding the rates of *tuples/second* over different window sizes, i.e. 5, 10 and 15 minutes, is presented in the Table below.

Throughput		
Window (min)	**System 1** [23]	**System 2** [23]
5	5,100 (t/s)	3,000 (t/s)
10	3,500 (t/s)	2,100 (t/s)
15	3,000 (t/s)	1,750 (t/s)

Table 5.1: *System S_1 and S_2 throughput*

Next, we moved on the performance comparison between the three most widely used algorithms, i.e. ScaleJoin, Handshake join and CellJoin algorithms. ScaleJoin's maximum number of comparisons per second increase linearly when executed in S_1. When all the 48 cores are used, ScaleJoin achieve 2.5 billion comparisons/sec more than Handshake join. As far as the performance of S_2 system, it seems that this system outperforms the Handshake join algorithm more than 1 billion comparisons/sec.

When it comes to ScaleGate performance, we see that throughput is not limited by the two ScaleGate instances at the articulation points of the architecture. When executed on S_2, ScaleGate's rate is increasing with the number of sources and does not degrade when more sources are added. More specifically, for a window size of 15 minutes ScaleGate's rate reaches 150,000 t/s which is 50 times higher of the highest throughput achieved. Finally when it comes to latency ScaleJoin achieves great results. Measurements have shown a maximum latency of 70 ms which is by far better than the original Handshake join which

can reach up to 7.5 minutes latency. ScaleJoin's latency increases linearly with the number of PTs. This is due to the fact that the more PTs available the lower the output rate per PT is.

5.4 Performance Evaluation

In this section we present the performance achieved by our system and compare the results with other state-of-the-art implementations. We also comment on our system's scalability when a single or four FPGAs are used as well as the resources utilization.

As we have already mentioned in chapter 4 our system is mapped on a Convey HC-2ex platform using four Virtex 6 LX760 FPGAs. Along with the FPGAs runs a 4-core Intel Xeon clocked at 2.13 GHz with 24 GB RAM. Our reconfigurable system runs at 150 MHz and uses close to 50% of the resources available. A more analytical display of the resources utilization is presented in the table below.

FPGA Available Resources	**System Utilization**
DSPs	0/864 (0%)
Slice Registers	281548/948480 (29%)
Slice LUTs	244328/474240 (51%)
Block RAMs	144/720 (20%)
IOBs	909/1200 (75%)

Table 5.2: *System Resources Utilization*

The software-based implementation as well as the first hardware implementation of ScaleJoin algorithm are going to be our reference as far as the comparisons that are going to be done. The previous hardware implementation is tested on the same platform as our proposed architecture. On the other hand, the original algorithm is tested on a system running a 48-core AMD Opteron at 2.6 GHz with 64 GB RAM. The performance was measured by evaluating the maximum number of comparisons that can be executed per second and the maximum throughput achieved, i.e. number of tuples per sec. The measurements are actual, experimental results from runs on the respective platforms.

5.4.1 HW vs. SW ScaleJoin Evaluation

This section compares the hardware-based implementation of the ScaleJoin algorithm vs. the best software-based solution. Figures 5.1 and 5.2 and Tables 5.3 and 5.4 present

the above mentioned measurements comparing our implementation with the software-based implementation.

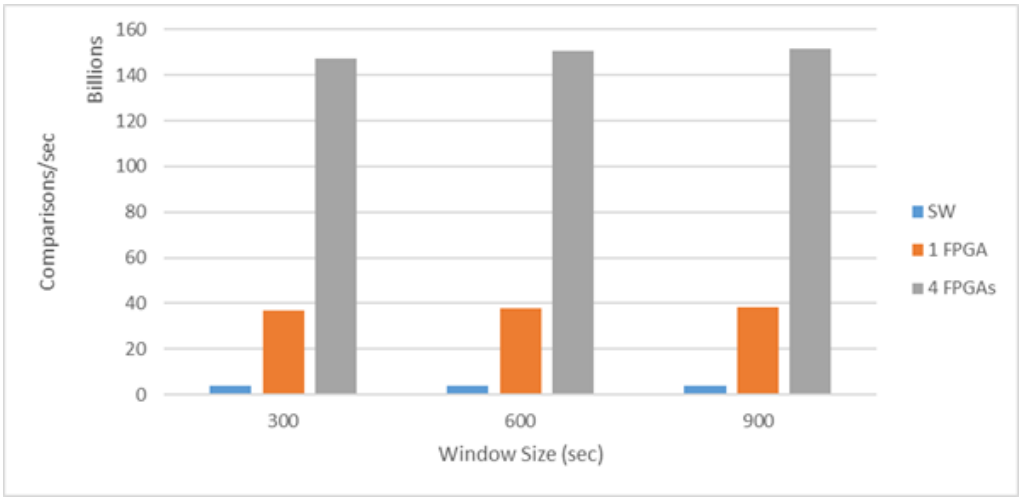


Figure 5.1: *Comparisons/sec for software-based multicore ScaleJoin and FPGA-based solution*

Comparisons/sec (c/s)			
Window (min)	**SW** [23]	**Proposed Architecture (1 FPGA)**	**Proposed Architecture (4 FPGA)**
5	4 billions	36.8 billions	147.5 billions
10	4 billions	37.6 billions	150.5 billions
15	4 billions	38.4 billions	151.8 billions

Table 5.3: *Comparisons/sec for software-based multicore ScaleJoin and FPGA-based solution*

As we can see our system outperforms the software-based multi-core solution by achieving $\times 9.5$ more comparisons per sec when a single FPGA is used and $\times 38$ more comparisons per sec when all four FPGAs are in use. To be more specific, the original software-based system carries out approximately 4 billion comparisons/sec while our system achieves approximately 38 billion comparisons/sec with a single FPGA and 152 billions comparisons/sec with all four FPGAs. From these results it becomes obvious that when it comes to comparisons per sec our system performance grows linearly with the number of FPGAs used.

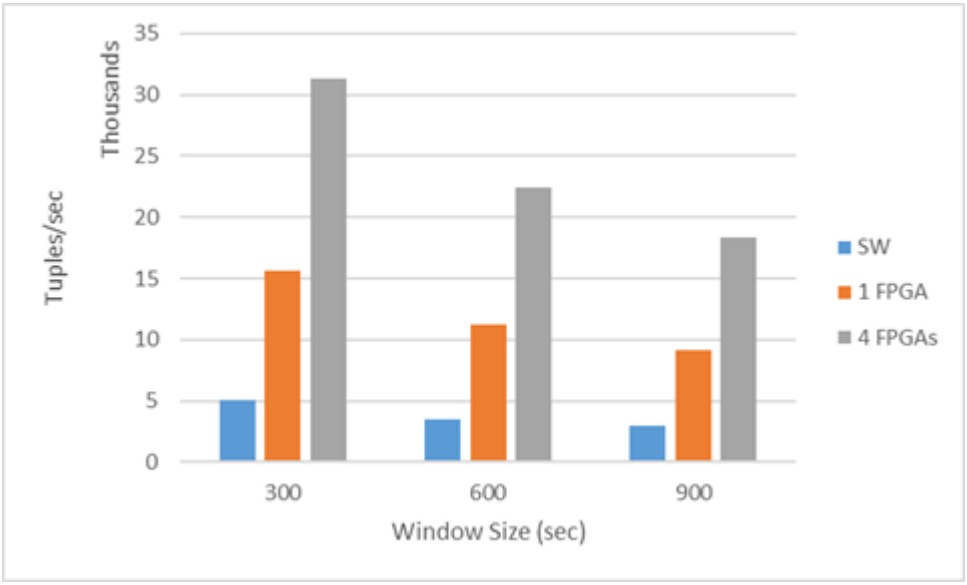


Figure 5.2: *Throughput (tuples/sec) for software-based multicore ScaleJoin and FPGA-based solution*

Throughput/sec (t/s)			
Window (min)	**SW** [23]	**Proposed Architecture (1 FPGA)**	**Proposed Architecture (4 FPGA)**
5	5100	15680	31360
10	3500	11200	22400
15	3000	9184	18368

Table 5.4: *Throughput (tuples/sec) for software-based multicore ScaleJoin and FPGA-based solution*

Regarding the throughput achieved in tuples per sec, our proposed architecture can handle a rate of approximately 9200 tuples per second when a single FPGA is used and 18400 tuples per second when all four FPGAs are used. This rate is at least $\times 6$ greater than the 3000 tupeles/sec of the software-based implementation. At this point we have to mention that our system scales proportionally to the square root of the number of total processing units. This can be easily understood from the formula above (Section 5.2) and it is also proven by the results in figure 5.2.

5.4.2 First HW Implementation vs. Proposed Architecture

This section compares our proposed architecture vs. the first hardware implementation of ScaleJoin algorithm. At this point we have to mention that both architectures are mapped in the same platform, i.e Convey HC-2ex. Figures 5.3 and 5.4 and Tables 5.5 and 5.6 present the results for both hardware implementations along with the software-based

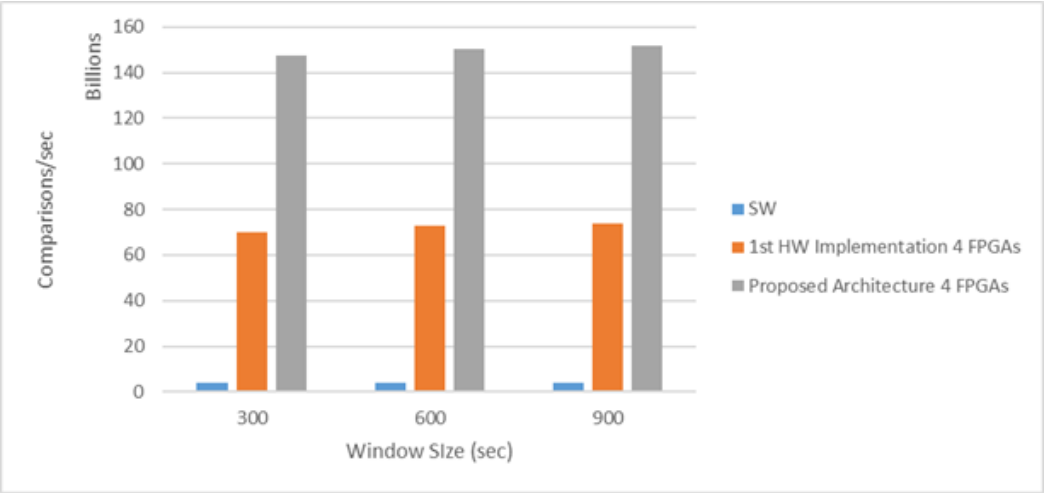


Figure 5.3: *Comparisons/sec for SW, 1st HW architecture and proposed architecture*

Comparisons/sec (c/s)			
Window (min)	**SW** [23]	**1st HW architecture (4 FPGA)**	**Proposed Architecture (4 FPGA)**
5	4 billions	70 billions	147.5 billions
10	4 billions	73 billions	150.5 billions
15	4 billions	74 billions	151.8 billions

Table 5.5: *Comparisons/sec for SW, 1st HW architecture and proposed architecture*

implementation as a reference. As we can see, our system achieves 2 times more comparisons/sec, which was expected because we managed to map a larger number of processing units, we achieved a greater level of parallelism and we made a better I/O management. Finally, when it comes to the maximum throughput rate (tuples/sec) our proposed architecture achieves 1.4 times greater throughput than the 1st hardware implementation.

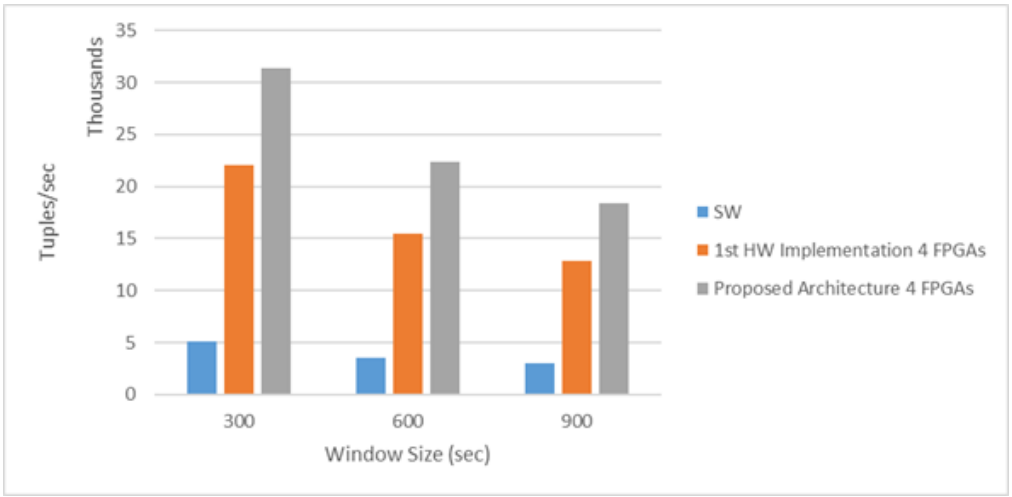


Figure 5.4: *Throughput (tuples/sec) for SW, 1st HW architecture and proposed architecture*

Throughput/sec (t/s)			
Window (min)	**SW** [23]	**1st HW architecture (4 FPGA)**	**Proposed Architecture (4 FPGA)**
5	5100	22000	31360
10	3500	15500	22400
15	3000	12800	18368

Table 5.6: *Throughput (tuples/sec) for SW, 1st HW architecture and proposed architecture*

5.4.3 HW-based Implementations vs. SW-based Algorithms

This section compares our HW-based Scalejoin implementation and the first hardware implementation of ScaleJoin algorithm with different previous software-based solutions with great performance results. Table 5.7 presents and compares the hardware-based implementations, with some of the state-of-the-art software implementations, which we described in section 2.3.1.

Systems	CPU Cores	CPU	Processing Time Window (mins)	Max Throughput Rate (tuples/sec)	Max Processing Rate (comps/sec)
Handshake Join	40	2.2 GHz AMD Opteron	15	3000	1.5 billion
ScaleJoin	48	2.6 GHz AMD Opteron	15	3000	4 billions
SplitJoin	32	4 × Inel E5-4650	15	5200	-
CellJoin	9	1 PPE and 8 SPEs	15	2000	1.5 billion
FPGA-based ScaleJoin	2 CPU + 4 FPGA	2.5 GHz Intel Xeon	15	12800	74 billions
Proposed Architecture	2 CPU + 4 FPGA	2.5 GHz Intel Xeon	15	18400	152 billions

Table 5.7: *Proposed architecture compared with other software and hardware implementations*

As we can see Handshake join and SplitJoin offer the best software solutions as far as the throughput rate (tuples/sec) with 5125 and 5200 tuples/sec, respectively. Our implementation though offers at least 3.5 times greater throughput and outperforms any other software-based solution. When we compare the number of comparisons per sec, we see that the best software solution is the original ScaleJoin algorithm, which manages to achieve approximately 4 billions comparisons per sec. When we compare that rate with our implementation, we observe that our architecture outperforms any other state-of-the art solution by carrying out a total of 152 billions comparisons.

5.5 Conclusions

In this section we summarize the results and the comparisons made in the above sections. As the results show we outperform the original ScaleJoin algorithm, by achieving at least 6 times greater throughput and manage to make 38 times more comparisons per sec. Our implementation, also, when compared with the best software-based solution, i.e SplitJoin, it achieves at least 3.5 times greater throughput. When we compare our proposed architecture with the first hardware implementation of Scalejoin algorithm, we see that our architecture achieves at least two times more comparisons per sec, which leads to a 1.4 times greater throughput. Finally, it is important to mention the scalability of the hardware proposed solution. As we showed in Figure 5.1 and Figure 5.2, the maximum number of comparisons per sec, scales linearly to the number of FPGAs used, while the maximum number of tuples per sec grows proportionally to the square root of the number of FPGAs.

Chapter 6

Conclusion

This work presented an FPGA-based system that implements stream join processing. Based on the original ScaleJoin algorithm as well as on its first hardware implementation we designed the proposed architecture which manages to improve the previous results. We adopted a different approach and managed to map a larger number of PUs, achieving greater level of parallelism. Our implementation, also, exploits the full bandwidth of Convey's HC-2ex memory controllers and almost doubles the utilization of the available resources. According to the results presented in chapter 5 our implementation is efficient and to the best of our knowledge outperforms any state-of-the-art work. Furthermore it is extensible, meaning it can be mapped to more than a single FPGA device - as we have already shown in chapter 5 - and scalable as the increasing number of PUs or FPGA devices offers better performance results. Finally, our proposed hardware-based architecture can be used as generic template for mapping stream processing algorithms to reconfigurable logic, taking into consideration real-world challenges and restrictions.

6.1 Future Work

Regarding future work, we are aware of certain adjustments that can significantly increase our processing throughput. As shown in Table 5.1, we use 51% of the available slice LUTs. This is because, the network of multiplexers used to forward the results, consumes a large amount of routing resources and restrict our potential of mapping a larger design. For this reason, a different logic should be used to forward the results, that will enable the mapping of more PUs. Finally, we could test our implementation on another platform and compare the results with our proposed architecture.

Bibliography

[1] http://www.sqlstream.com/blog/2014/05/sqlstream-plus-apache-kafka/.

[2] Apache samza. http://samza.incubator.apache.org/.

[3] Vowpal wabbit (vw), 2007. http://hunch.net/ vw.

[4] Storm, 2011. http://storm-project.net.

[5] Huawei, 2015. http://huawei-noah.github.io/streamDM/.

[6] Bifet A., Holmes G., Kirkby R., and Pfahringer B. Data stream mining - a practical approach. *MOA*, pages 107–139, 2011.

[7] D. J. Abadi. The design of the borealis stream processing engine. *CIDR*, 2005.

[8] S. Agarwal and B. R. Prasad. High speed streaming data analysis of web generated log streams. *IEEE 10th International Conference on Industrial and Information Systems*, pages 413–418, 2015.

[9] Babcock B. and Olston C. Distributed top-k monitoring. In *ACM SIGMOD International Conference on Management of Data*, 2003.

[10] B. Brian, M. Data, and R. Motwani. Load shedding for aggregation queries over data streams. *Data Engineering Proceeding. 20th International Conference on IEEE*, 2004.

[11] Aggarwal C. A framework for diagnosing changes in evolving data streams. *ACM SIGMOD Conference*, 2003.

[12] Aggarwal C., Han J., Wang J., and Yu P. A framework for clustering evolving data streams. *VLDB Conference*, 2003.

[13] Aggarwal C., Han J., Wang J., and Yu P. On-demand classification of data streams. *ACM KDD Conference*, 2004.

[14] Giannella C., Han J., Pei J., Yan X., and Yu P. Mining frequent patterns in data streams at multiple time granularities. In *NSF Workshop on Next Generation Data Mining*, 2002.

[15] Intanagonwiwat C., Govindan R., and Estrin D. Directed diffusion: a scalable and robust communication paradigmfor sensor networks. In *6th ACM International Conference on Mobile Computing and Networking*, 2000.

[16] Kifer D., David S.-B., and Gehrke J. Detecting change in data streams. *VLDB Conference*, 2004.

[17] J. A. Daniel. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, 2003.

[18] Dong G., Han J., Lam J., Pei J., and Wang K. Mining multi-dimensional constrained gradients in data cubes. In *VLDB*, 2001.

[19] Hulten G., Spencer L., and Domingos P. Mining time changing data streams. *ACM KDD Conference*, 2001.

[20] J. Gama. Knowledge discovery from data streams. *Chapman & Hall/CRC*, 2010.

[21] Bugra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 2009.

[22] Bart Goethals and Mohammed J. Zaki. Workshop report on workshop on frequent itemset mining implementations. *FIMI*, 2003.

[23] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. Scalejoin: A deterministic, disjoint-parallel and skew-resilient stream join. *Big Data (Big Data), 2015 IEEE International Conference*, 2015.

[24] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conference on Management of Data*, 2000.

[25] Jun Huan, Wei Wang, Deepak Bandyopadhyay, Jack Snoeyink, Jan Prins, and Alexander Tropsha. Mining protein family-specific residue packing patterns from protein structure graphs. *Eighth International Conference on Research in Computational Molecular Biology (RECOMB)*, pages 308–315, 2004.

[26] Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. An apriori-based algorithm for mining frequent substructures from graph data. *PKDD 2000*, pages 13–23, 2000.

[27] Aslam J., Butler Z., Constantin F., Crespi V., Cybenko G., and Rus D. Tracking a moving object with a binary sensor network. In *ACM SenSys*, 2003.

[28] Ruoming Jin and Gagan Agrawal. A systematic approach for optimizing complex mining tasks on multiple datasets. In *ICDE*, 2005.

[29] Charalabos Kritikakis, Grigorios Chrysos, and Apostolos Dollas. An fpga-based high-throughput stream join architecture. *Field Programmable Logic and Applications (FPL), 2016 26th International Conference*, 2016.

[30] Michihiro Kuramochi and George Karypis. Frequent subgraph discovery. In *IEEE International Conference on Data Mining,*, page 313–320, 2001.

[31] Neumeyer L., Robbins B., Nair A., and Kesari A. S4: Distributed stream computing platform. In *ICDMW, IEEE Press*, pages 170–177, 2010.

[32] Datar M., Gionis A., Indyk P., and Motwani R. Maintaining stream statistics over sliding windows. *SIAM Journal on Computing*, page 1794– 1813, 2002.

[33] S. Murthy, N. Tony, B. Avalani, X. Wang, K. Wang, and A. Gangadharan. Pulsar-real-time analytics at scale. *eBay Inc*, 2015.

[34] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Splitjoin: a scalable, low-latency stream join architecture with adjustable ordering precision. *USENIX ATC '16 Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, 2016.

[35] Yasin Oge, Takefumi Miyoshi, and Hideyuki Kawashima. Design and implementation of a merging network architecture for handshake join operator on fpga. *Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium*, 2012.

[36] Domingos P. and Hulten G. Mining high-speed data streams. In *Proceedings of the ACM KDD Conference*, 2000.

[37] B. R. Prasad and Agarwal S. Handling big data stream analytics using samoa framework - a practical experience. *Database Theory and Application*, pages 197–208, 2014.

[38] Agrawal R., Imielinski T., and Swami A. Mining association rules between sets of items in large databases. *ACM SIGMOD Conference*, 1993.

[39] Jin R. and Agrawal G. An algorithm for in-core frequent itemset mining on streaming data. *ICDM Conference*, 2005.

[40] Prasad B. R. and Agarwal S. Critical parameter analysis of vertical hoeffding tree for optimized performance using samoa. *Mach. Learning & Cybern*, 2016.

[41] Pratanu Roy, Jens Teubner, and Rainer Gemulla. Low-latency handshake join. *Proceedings of the VLDB Endowment*, 2014.

[42] Guha S., Mishra N., Motwani R., and O'Callaghan L. Clustering data streams. *IEEE FOCS Conference*, 2000.

[43] Mathew S. Overview of amazon web services. *Amazon White Papers*, jan 2014.

[44] Dasu T., Krishnan S., Venkatasubramaniam S., and Yi K. An information-theoretic approach to detecting changes in multidimensional data streams. Technical report, Duke University, 2005.

[45] Jens Teubner and Rene Mueller. How soccer players would do stream joins. *SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[46] Chen Y., Dong G., Han J., Wah B. W., and Wang J. Multi-dimensional regression analysis of time-series data streams. In *VLDB*, 2002.

[47] M. Zaharia, C. Mosharaf, M. J. Franklin, S. Shenker, and S. Ion. Spark: cluster computing with working sets. In *2nd USENIX conference on Hot topics in cloud computing*, 2010.