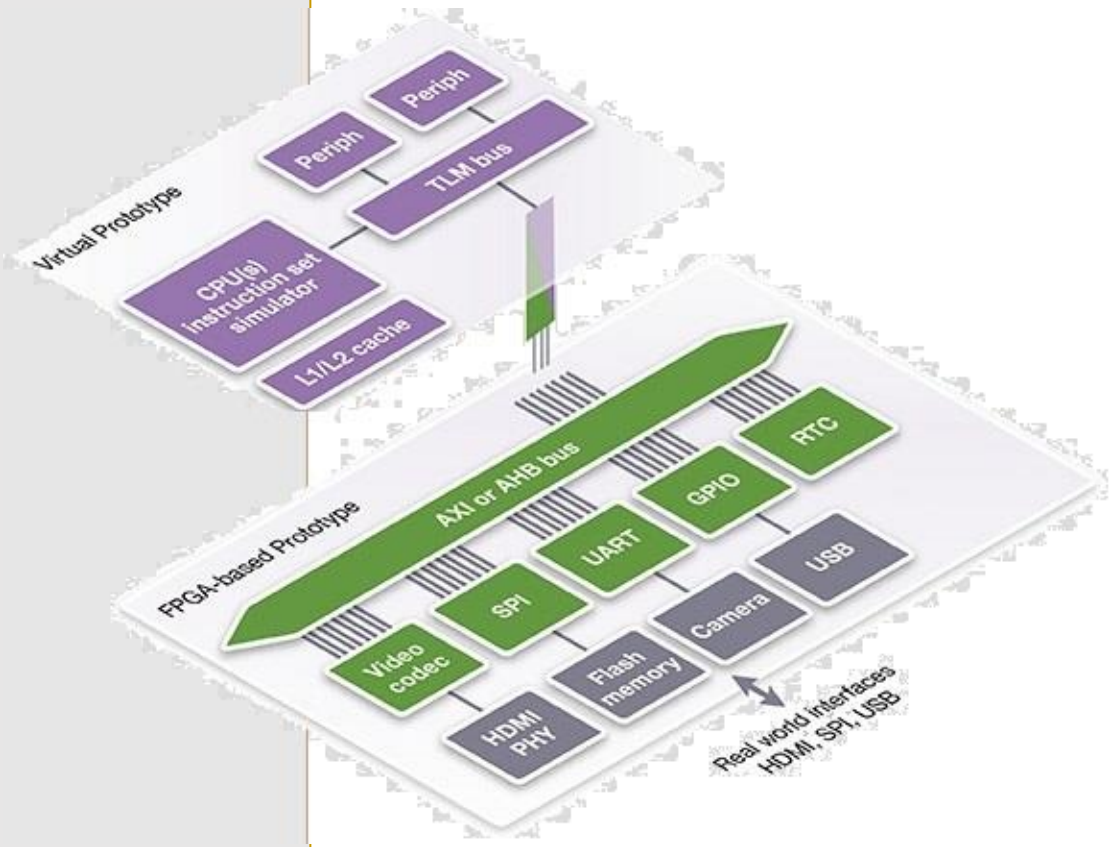
Platform Based on Reconfigurable Computing

Σχεδίαση Πλατφόρμας Βασισμένης σε Αναδιατασσόμενη Λογική για Εξόρυξη Δεδομένων από Ροές

Kritikakis Charalampos / Κρητικάκης Χαράλαμπος

Technical University of Crete



Examination Committee

Apostolos Dollas (Supervisor)
Dionisios Pneumatikatos
Minos Garofalakis

Chania, May 2016

Abstract

Stream join is one of the most fundamental operations to relate information from different streams. The challenging task is to use systems with memory and run-time constraints in order to store and analyze massive volumes of streaming data. Hence, it is important to maximize the processing which is done on-the-fly as the streaming data arrives. This paper presents an FPGA-based architecture that maps the most performance-efficient stream join algorithm, i.e. ScaleJoin, to reconfigurable logic. The system was fully implemented on a Convey HC-2ex hybrid computer and experimental performance evaluation shows that the proposed system outperforms by up to one order of magnitude the corresponding fully optimized software-based solution running on a high-end 48-core multiprocessor platform, by exploiting the high internal bandwidth of present-day FPGAs. The proposed architecture can be used as a generic template for mapping other stream processing algorithms, taking into consideration real-world challenges.

Ευχαριστίες

Οι ευχαριστίες είναι σιγουρά ένα από τα πιο δύσκολα κομμάτια, αν όχι το πιο δύσκολο. Πλέον, κλείνει ένα μεγάλο κεφάλαιο στη ζωή μου. Αυτό είναι η προπτυχιακή μου φοίτηση, που για τους περισσοτέρους φοιτητές είναι το πρώτο ή ένα από τα πρώτα «κατορθώματα» τους. Συνεπώς, όπως είναι φυσικό, θα ήθελα να ευχαριστήσω κάποια άτομα που με βοήθησαν στο να πέτυχω τον σκοπό μου, ο οποίος είναι το πτυχίο.

Αρχικά, θα ήθελα να ευχαριστήσω τον Καθηγητή Απόστολο Δόλλα, ο οποίος είναι ο επιβλέπων της διπλωματικής μου, για την καθοδήγηση, το ενδιαφέρον που έδειξε σε εμένα και δείχνει σε όλους τους φοιτητές, πράγμα αξιοθαύμαστο, και την προθυμία του να βοηθήσει σε οποιαδήποτε στιγμή. Νοιώθω ιδιαίτερη τιμή που τον έχω καθηγητή και ακόμα μεγαλύτερη που είναι ο επιβλέπων μου. Επίσης, νοιώθω ιδιαίτερα τυχερός που τον γνώρισα, διότι χωρίς την βοήθεια του δεν θα είχα αυτά τα αποτελέσματα και επίσης για τις συμβουλές που μου έδωσε για την καλύτερη πορεία μου στο μέλλον. Αν έπρεπε να πω ένα πράγμα που θα έδινε μια γενική εικόνα για τον κύριο Δόλλα, αυτό είναι ότι είναι κάτι παραπάνω από ένας εξαιρετικός καθηγητής και επιστήμονας. Και πάλι τον ευχαριστώ θερμά για όσα μου πρόσφερε.

Επίσης, θα ήθελα να ευχαριστήσω ιδιαίτερα όλο το προσωπικό του MHL. Ξεκινώντας από τους υπολοίπους διδάσκοντες, Καθηγητή Διονύση Πνευματικάτο και Αναπληρωτή Καθηγητή Γιάννη Παπαευσταθίου, που ήταν πάντα πρόθυμοι να βοηθήσουν, έκαναν πραγματικά ένα εξαιρετικό μάθημα και ταυτόχρονα ευχάριστο στην παρακολούθηση και αυτά συνέβαλαν σημαντικά στην επιλογή του τομέα που ακολούθησα. Ακόμα, θα ήθελα να ευχαριστήσω το ακαδημαϊκό προσωπικό του MHL. Ξεκινώντας από τους υπευθύνους του εργαστήριου, κύριο Ευριπίδη Σωτηριάδη, Κυπριανό Παπαδημητρίου και Μάρκο Κιμιωνη για την εξαιρετική διεξαγωγή των εργαστήριων και την βοήθεια τους. Τέλος, θα ήθελα να ευχαριστήσω ιδιαίτερα τους Ιωσήφ Κοϊδη, Δημήτρη Θεοδωρόπουλο και Γιώργο Χαριτοπουλο για τις ατέλειωτες ώρες που πέρασα στο γραφείο τους, λόγω διπλωματικής, με τον Γρηγόρη.

Επιπλεον, θα ήθελα να ευχαριστήσω τον φίλο μου, Γρηγόρη Χρυσό, για την στήριξη και καθοδήγηση του στο τελευταίο και πιο βασικό έτος της φοίτησής μου. Η βοήθεια που πρόσφερε στη διπλωματική μου, σε ένα αρκετά δύσκολο κομμάτι της σύγχρονης Επιστήμης των Υπολογιστών, ήταν αξιοθαύμαστη. Τον ευχαριστώ για τις ατέλειωτες ώρες που σπάγαμε το κεφάλι μας να δούμε τι «πάει στραβά», τις ευχάριστες στιγμές του debug στην Convey (πλακά κάνω) και τις γνώσεις που μου πρόσφερε για την προσέγγιση προβλημάτων, αντίστοιχων της διπλωματικής μου. Πραγματικά όλα θα ήταν τελείως διαφορετικά αν δεν είχαμε εργαστεί μαζί και τελικώς και ελπίζω ότι δικαιωθήκαμε από το αποτέλεσμα.

Ακόμα, θα ήθελα να ευχαριστήσω τον Καθηγητή Μίνο Γαροφαλάκη και Καθηγητή Διονύση Πνευματικάτο για την συμμετοχή τους στην εξεταστική μου επιτροπή και την μελέτη της παρούσας εργασίας.

Φεύγοντας από τα άτομα του Πολυτεχνείου Κρήτης, θα ήθελα να ευχαριστήσω και τους φίλους μου. Ξεκινώντας από τους συμφοιτητές μου, θα ήθελα να ευχαριστήσω τους φίλους μου, Πέτρο, Θοδωρή, Bober, Μιχάλη, Μιχάλη, Γιώργο και Πολίτση, για την αλληλοβοήθεια,

καθοδήγηση, στήριξη και τις άπειρες στιγμές γέλιου. Είναι πραγματικά εκπληκτικοί, τους ευχαριστώ για όλα και πιστεύω ότι με όλα όσα έχουμε περάσει έχουμε θέσει τις βάσεις για μακροχρόνιες φίλιες. Μακάρι να μπορούσα να γράψω για όλους του φίλους - συμφοιτητές κάτι αλλά θα χρειαζόμουν πολλές σελίδες ακόμα.
Με το να σπουδάσω στην πόλη μου, δεν χάθηκα και με τους παιδικούς μου φίλους από Χανιά. Συνεπώς, θα ήθελα να ευχαριστήσω τους Τσακίρη, Νταμα, Μάνο, Σταμάτη, Νίκο, Ευτύχη, Αλεξάνδρα, Ρένα και Ελένη που με έκαναν να ξεχνιέμαι από τα προβλήματα που είχα στην σχολή, ενώ ταυτόχρονα μου προσφέραν άπειρες ευχάριστες στιγμές και αναμνήσεις. Πηγαίνοντας προς Ρέθυμνο, δεν μπορώ να μην ευχαριστήσω τις φίλες μου Μαρία και Ζωή για όλες τις φορές που με φιλοξενήσανε για να ξεφύγω από την καθημερινότητα και περάσαμε αμέτρητες στιγμές μαζί.

Σε αυτό το σημείο, θα ήθελα να πω ένα μεγάλο ευχαριστώ στην κοπέλα μου, Δέσποινα, με την οποία πέρασα τα περισσότερα και πιο δύσκολα χρόνια της φοίτησης μου, για την καθημερινή στήριξη στις δυσκολίες που πέρασα, τα γέλια και τις στιγμές που με έκανε να ξεχνιέμαι, έτσι ώστε να συνεχίσω να δουλεύω. Θεωρώ αξιοθαύμαστη την επιμονή της και την υπομονή της, όντας σε τόσο διαφορετικές σχολές, με διαφορετικό φόρτο και πιστεύω ότι δεν έχουν πολλοί την τύχη που είχα εγώ να την γνωρίσω.

Τέλος, θα ήθελα να ευχαριστήσω την όλη οικογένεια μου, πιο συγκεκριμένα τους γονείς μου, Γιώργο και Μαρία και τις αδερφές μου Φωτεινή και Κυριακή, για την ψυχολογική στήριξη τους σε όλη την διάρκεια της φοίτησης μου και που μου προσφέραν ένα ιδανικό περιβάλλον εργασίας, κατά την διάρκεια των σπουδών μου. Ακόμα, ένα τεράστιο ευχαριστώ και στον θείο μου, Γιάννη, για όλα τα χρόνια που με στήριξε, που δουλέψαμε μαζί και που ήταν πάντα κάτι σαν το μεγάλο μου αδερφό.

Table of Contents

Abstract 3
Ευχαριστίες 5
Table of Contents 7
Table of Figures and Tables 10
Chapter 1: Introduction 11
1.1 Introduction 11
1.2 Terminology 11
1.2.1 Big Data 11
1.2.2 Data Mining 12
1.2.3 Streams 13
1.2.4 Stream Data Mining 13
1.3 Scientific contribution 14
1.4 Structure of this thesis 15
CHAPTER 2: Related Work 16
2.1 Introduction 16
2.2 Streaming Data Mining Types and Algorithms 16
2.3 Distributed Stream Data Mining Platform: Samoa 18
2.3.1 What is SAMOA? 18
2.3.2 Why SAMOA? 19
2.3.3 Samoa's Design Goals 20
2.3.4 Algorithms used in SAMOA 21
2.4 Stream Join Processing Implementations 21
2.4.1 Software based implementations 22
2.4.2 Hardware based implementations 22
Chapter 3: Algorithm Analysis 24
3.1: Introduction 24
3.2: Algorithm's terminology 24
3.2.1 Ready tuple 24
3.2.2 Tuple and Time based Windows 25
3.2.3 Tuple Merge 25
3.3: The ScaleJoin algorithm 25
3.3.1 Problems and challenges 25
3.3.2 Parallelization 26
3.3.3 Abstract Data Type: ScaleGate 26

3.3.4 ScaleJoin Basic Steps 27
3.4: ScaleJoin evaluation 27
3.5: ScaleJoin analysis 28
3.5.1 ScaleGate input (SG_{in}) 29
3.5.2 Processing units 30
3.5.2 ScaleGate output (SG_{out}) 30
3.6 Thoughts and processing steps 31
3.7 Conclusion 33
Chapter 4: ScaleJoin Architecture 34
4.1 Introduction 34
4.2 Convey HC-2 34
4.2.1 HC-2 server 34
4.2.2 Hc-1 system architecture 35
4.2.3 Intel host processor 35
4.2.4 Convey FPGA-based coprocessor 36
4.2.5 Programming model 37
4.2.6 Porting already existing applications 38
4.2.7 The Convey Personality Development Kit 39
4.2.8 AE-to-AE Interface 39
4.2.9 Management/Debug Interface 40
4.3 Reconfigurable ScaleJoin System 41
4.2 Scalejoin's Implementation 42
4.3.1 System input (SGin) 45
4.3.2 ScaleJoin module 47
4.3.3 System's output (SGout) 49
4.3.4 Scalejoin's Control 50
4.4 Comparison modules 51
4.4.1 Comparison module architecture 51
4.4.2 Process control module 53
4.4.3 Parallelization of processing units 54
4.5 Processing Unit 56
4.6 Conclusion 57
Chapter 5: Evaluation 58
5.1 Introduction 58
5.2 Cross-check evaluation 58

5.3 Theoretical Performance bounds 58
5.4 Experimental setup 59
5.5 Performance Evaluation 59
5.6 Benchmark Performance Evaluation 62
5.7 Conclusion 63
Chapter 6: Conclusion and Future work 64
6.1 Conclusion 64
6.2 Future work 64
References 66

Table of Figures and Tables

Table 2.1 ML frameworks 20
Figure 2.2: Samoa's Inner Architecture 21
Table 3.1: ScaleJoin basic Steps 27
Table 3.2 ScaleJoin evaluation table 28
Figure.3.2: Overview of Scalejoin's architecture 29
Figure 3.3: PU flowchart 31
Figure 3.4 Correlation Graph 32
Figure 4.1 HC-1 server architecture. 35
Figure 4.2 HC-1 Architecture. 36
Figure 4.3 Coprocessor three main components AEH, AEs, MCs. 36
Figure 4.4 HC-2 runtime environment 38
Figure 4.5 AE-to-AE Interface Diagram 40
Figure 4.6 Management Interface Diagram. 40
Figure 4.7 FPGA-based Architecture 42
Figure 4.8: ScaleJoin architecture 44
Figure 4.9: ScaleGate in Architecture 46
Figure 4.10: Comparison module 48
Figure 4.11: ScaleGate out 49
Figure 4.12: ScaleJoin Control 50
Figure 4.13: Comparison module architecture 52
Figure 4.14: Comparison Control 54
Figure 4.15: Reconfigurable StreamJoin Architecture 55
Figure 4.16: PU architecture 57
Table 5.1: Resource utilization for hardware-based stream join architecture 60
Figure 5.1: Processing rate (comparisons/sec) for SW-based multicore ScaleJoin and FPGA-based solution ScaleJoin 61
Figure 5.2: Throughput (tuples/sec) for SW-based multicore ScaleJoin and FPGA-based solution ScaleJoin 61
Table 5.2. SW multicore stream join vs. FPGA based stream join on Benchmark Evaluation 63

Chapter 1: Introduction

1.1 Introduction

In this chapter of this Thesis, we make reference to the bibliography of Stream Data Mining or Stream Mining. Stream Data Mining is a new research subject in Computer Science. Stream data mining is a combination of the research in Data Mining and Streams and it has been studied extensively because of the volume of data in many applications. So, converting this big amount of data into useful information and knowledge is a fundamental process and an area worth studying. Before we analyze what Stream Data Mining is, we will make reference to Big Data, Streams and Data Mining.

1.2 Terminology

1.2.1 Big Data

Big data is a recent term that has appeared to define the large amount of data that surpasses the traditional storage and processing requirements. Volume, Velocity and Variety, also called the 3' Vs, is commonly used to characterize big data. Looking at each of the 3'Vs independently brings challenges to big data analysis.

Big Data 3Vs

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 by using data warehousing techniques or using parallel processing architecture systems.

Velocity: Velocity deals with the rate in which data is generated and flows into a system. Everyday sensor devices and 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. In aiming to process these streams of unbounded flow of data, some frameworks have emerged like the Apache! S4.

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.

1.2.2 Data Mining

Data mining is the process of extraction of previously unknown and potentially useful information from raw data [1]. Despite data mining being a relatively new term, this field has been widely used by various organizations, i.e. financial, communication, and marketing companies. Data mining algorithms need to explore and analyze huge volumes of data under strict space and time restrictions in order to enhance the value of existing information resources. Modern applications, which require real-time processing of high-volume data streams, are pushing the limits of traditional data processing infrastructures [2]. Thus, as the data mining research field is evolving, the researchers focus on new concepts and trends, like the scalability of the proposed algorithms and systems, the implementation of performance efficient distributed systems and the development of real time data mining solutions.

Use of Data Mining

Data mining is primarily used today by companies with a strong consumer focus - retail, financial, communication, and marketing organizations. It enables these companies to determine relationships among factors such as price, product positioning, or staff skills, and "external" factors such as economic indicators, competition, and customer demographics. It enables them to determine the impact on sales, customer satisfaction, and corporate profits. Finally, it enables them to "drill down" into summary information to view detail transactional data. With data mining, a retailer could use point-of-sale records of customer purchases to send targeted promotions based on an individual's purchase history. By mining demographic data from comment or warranty cards, the retailer could develop products and promotions to appeal to specific customer segments.

Data Mining Tasks

Data mining involves six common classes of tasks:

- Anomaly detection: The identification of unusual data records, that might be interesting or data errors that require further investigation.
- Association rule learning (Dependency modelling): Searches for relationships between variables. For example a supermarket might gather data on customer purchasing habits. Using association rule learning, the supermarket can determine which products are frequently bought together and use this information for marketing purposes. This is sometimes referred to as market basket analysis.

- Clustering is the task of discovering groups and structures in the data that are in some way or another "similar", without using known structures in the data.
- Classification is the task of generalizing known structure to apply to new data. For example, an e-mail program might attempt to classify an e-mail as "legitimate" or as "spam".
- Regression attempts to find a function which models the data with the least error.
- Summarization: providing a more compact representation of the data set, including visualization and report generation.

1.2.3 Streams

In computer science, a stream is a sequence of data elements made available over time. A stream can be thought of as items on a conveyor belt 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 and formally, streams are potentially unlimited, not like data which is finite. Functions that operate on a stream, producing another stream, are known as filters, and can be connected in pipelines, analogously to function composition. Filters may operate on one item of a stream at a time, or may base an item of output on multiple items of input, such as a moving average.

The limitations and the drawbacks of the traditional data mining systems for big data collections lead to the need for in-stream processing. Stream processing solutions focus on continuous computations that take place as data flows in the system. The stream processing systems are designed to process high-volume data with low latency and in a scalable way.

1.2.4 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 subfield 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. Often, concepts from the field of incremental learning, a generalization of Incremental heuristic search are applied to cope with structural changes, on-line learning and real-time demands. 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, may change over time. This problem is referred to as concept drift.

As one can easily understand is that processing a 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. Examples of these challenges and adjustments are the input of the algorithm, which in Streams is infinite in comparison with Data Mining that data are finite and the configured algorithm must have a real-time response. In the Related work Chapter we make a more analytical reference to the problems that Stream Data Mining algorithms have and discuss the challenges that each type of algorithm referred has.

A fundamental operator for the data stream mining is the stream join operator [3]. 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 [4]. The stream join operator is computationally expensive [5] 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 [6, 7, 8] and other works that use hardware-based solutions [9 - 15]. Thus, The ScaleJoin algorithm [6] 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 input streams and it guarantees deterministic processing with scalable and high-throughput parallelism. The main core of the ScaleJoin algorithm is an abstract data-type, i.e. the ScaleGate, which is presented in more details in Chapter 3.

1.3 Scientific contribution

The contributions of this work are:

- This is the first hardware-based work, to the best of our knowledge, which proposes a reconfigurable architecture for the ScaleJoin stream join algorithm that is considered the most performance efficient state-of-the-art stream join technique.
- The proposed hardware-based architecture is scalable and generic. This architecture can be used for many streaming problems that need to correlate streaming data. Also, this is the first, to the best of our knowledge, hardware-based work that is not restricted by the hardware resources due to the size of the processing time window.
- The proposed architecture is extensible, as it takes advantage of the fine-grained and the coarse-grained parallelism that reconfigurable hardware can offer. In more details, this is the first presented work that takes advantage of a multi-FPGA platform for mapping the stream join operator.

- The implemented system achieves at least 4x better throughput data rates vs. the fastest stream join multi-threaded solution, when it is mapped on a high-end multi-processor platform. Also, we show that the proposed solution can offer at least one order of magnitude higher processing rates than any other multi-core published solution.

1.4 Structure of this thesis

This thesis is organized as follows. Chapter 2 presents other stream mining types and platforms and briefly compares other works, implementations and algorithms, regarding stream joins and makes an introduction and analyzes some preliminary concepts on the stream join problem. Section 3 describes the ScaleJoin algorithm, its new abstract data type and its challenges. Section 4 presents our proposed architecture and explains how it meets the requirements of the ScaleJoin algorithm. Section 5 evaluates the performance of the proposed architecture on a multi-FPGA platform and compares the performance results with the performance of previously presented works. Section 6 draws the conclusions of this work.

CHAPTER 2: Related Work

2.1 Introduction

In this Chapter, we address some of the current problems in the bibliography of Stream Data Mining, which is a quite new research chapter in Computer Science. We also present a new Stream Data Mining platform, called SAMOA, developed in Yahoo! Labs Barcelona, which addresses the main problems in Stream Mining and provides solutions to them. Finally, we present general stuff as it comes to Join processing and we make a reference to the studies that have been made on the algorithms that are compared with ScaleJoin algorithm in [6], both Hardware based and Software based.

2.2 Streaming Data Mining Types and Algorithms

A huge part of Computer Science division focuses on Data Mining, Machine learning and Streams in the last decade. So, besides Join processing, in Steam Data Mining many other problems and challenges associated with them exist and for each problem many algorithms exist to have them solved, but what changes is how they process the data. Therefore, in this point, we present to you the most common problems, as it comes to Stream data mining and we make a reference to the most common algorithms for each type of problem.

Data Stream Clustering Algorithms: Clustering is a widely studied problem in the data mining literature. However, it is more complicated when it comes to adapting 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 [16] 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 sets.
Here, it is worth mentioning the technique of micro-clusters [17] that defines clusters over the entire data set. In order to apply this technique to a variety of data mining algorithms, a micro-clustering based stream mining framework is utilized. This framework is designed by a summary of information which is defined by Micro-clusters and Pyramidal Time Frame data structures. The Micro-clusters technique can be extended to the problem of Data Stream Classification.

Data Stream Classification Algorithms: 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, to consider an algorithms as effective, it needs to be designed to take temporal locality into account. Also typical problems, which must be taken into account in classification are the memory needed to implement the techniques of

classification, the concept of Drifting in which sends the result of the classifier over time and the reliability of the classification.
Classic example of Stream Classification algorithms is Very Fast Decision Trees algorithm (VFDT) or Hoeffding Trees developed by Domingos and Hulten [35] 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 [20] in On-Demand classification in [17]. 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.

Frequent Pattern/Itemset Mining: The problem of frequent pattern mining was first introduced in [21], and was extensively analyzed for the typical 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 (either time or tuple based) or in an entire data stream [22, 23].
Manku and Motwani proposed the first one-pass algorithm, Lossy Counting, to find all frequent itemsets over an entire data stream [24]. This algorithm is a false-positive oriented in the sense that it does not allow false negatives, and has a provable bound on false positives. It uses a user-defined error parameter ϱ to control the quality of the answering set for a given support level θ. As a closure, a recent work by Karp, Papadimitriou and Shenker on finding frequent elements (or 1-itemset) [25]. Formally, given a sequence of length N and a threshold θ ($0 < \theta < 1$), the goal of their work is to determine the elements that occur with frequency greater than Nθ.

Change Detection in Data Streams: The patterns in a data stream may evolve over time. In many cases, it is desirable to track and analyze the nature of these changes over time. In [26, 27, 24], a number of methods have been discussed for change detection of data streams. In addition, data stream evolution can also affect the behavior of the underlying data mining algorithms when it comes to the results that become stale over time.
A known change detection method is the Velocity Density Method. The idea in velocity density is to construct a density based velocity profile of the data. This is analogous to the concept of kernel density estimation in static data sets. Kernel density estimation [28], provides a continuous estimate of the density of the data at a given point. The value of the density at a given point is estimated as the sum of the smoothed values of kernel functions K'h (·) associated with each point in the data set. Each kernel function is associated with a kernel width h which determines the level of smoothing created by the function.

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 [29,30]. Therefore, an interesting cube stream architecture that performs efficiently on-line partial grouping of current multi- dimensional data capture the basic dynamic and evolving characteristics of data streams, and facilitates fast OLAP data stream . Power cube architecture facilitates online analytical processing of data streams. It is also a preliminary structure for online mining stream.
As a reference to Stream Cube Analysis , we propose two methods, which are popular-path cubing, which rolls up the cuboids from the m-layer to the o-layer, by following the most popular drilling path, materializes only the layers along the path, and leaves other layers to be computed at OLAP query time and Popular-path-based stream cube computation Computing initial stream cube, i.e., the cuboids along the popular-path between the m-layer and the o-layer, based on the currently collected set of input stream data.

Dimensionality Reduction and Forecasting in Data Streams: Because of the inherent temporal nature of data streams, the problems of dimensionality reduction and forecasting is particularly important. When there are a large number of simultaneous data stream, we can use the correlations between different data streams in order to make effective predictions [31, 32] on the future behavior of the data stream. In particular, the well-known MUSCLES method [32] has been discussed, and its application to data streams have been explored. In addition, MUSCLES tries to predict the value of a Stream, under the previous values of all Streams and the current values from other streams.
Furthermore, SPIRIT algorithm exists, which explores the relationship between dimensionality reduction and forecasting in data streams. More precisely, SPIRIT operates on the column-vectors of observed stream values x_t and continually updates the participation weights. SPIRIT also adapts the number k of hidden variables necessary to capture most of the information. The adaptation is performed so that the approximation achieves a desired mean-square error.

2.3 Distributed Stream Data Mining Platform: Samoa

In this section, we present to you Samoa [33], the basic information about it, why Samoa was developed and Samoa's inner architecture.

2.3.1 What is SAMOA?

SAMOA is a tool to perform mining on big data streams. It is basically a distributed streaming machine learning (ML) framework, for example it is a Mahout but for stream mining. SAMOA contains a programing abstraction for distributed streaming

ML algorithms to enable development of new ML algorithms without dealing with the complexity of underlying streaming processing engines (SPE, such as Twitter Storm and S4). SAMOA also provides extensibility in integrating new SPEs into the framework. These features allow SAMOA users to develop distributed streaming ML algorithms once and they can execute the algorithms in multiple SPEs, i.e. code the algorithms once and execute them in multiple SPEs.

2.3.2 Why SAMOA?

Big Data is always evolving and one of the ways to mine big data is by using the streaming ML paradigm. This paradigm implies that the corresponding ML model for data mining will utilize real-time feedback and ML model updates will be faster. The ML model will adapt to changes via concept drift to handle adversarial interactions (such as spam) with the ML model. The main usage of streaming ML paradigm is to provide immediate feedback to user based on certain actions.

Concrete example of big data stream mining is spam detection on Yahoo! News or Yahoo! Mail. The spams' characteristics change over time, hence we need to retrain the ML model with new arriving data. Moreover, we also need to quickly develop new ML algorithms for big data stream mining.

The following are the existing solutions that are not designed for big Data Streams:

- **Mahout** is suitable for batch processing of the data and it is not designed for stream machine learning
- **Massive Online Analysis** (MOA) is suitable for stream machine learning, but it does not scale. It is only able to execute on a single machine and it could no be scaled into multiple machines if needed.

Furthermore, there is an existing solution called Jubatus, which is a distributed machine learning framework for big data stream. However, Jubatus implementation is tightly coupled between distributed ML algorithms and the underlying distributed streaming computation platform.

Table below summarizes existing machine learning framework and their characteristics related to mining big data streams.

Table 2.1 ML frameworks

MACHINE LEARNING FRAMEWORK	HORIZONTALLY SCALE	SPEED/REAL-TIME ANALYTICS	LOOSE-COUPLING
Mahout	Yes	No	No
MOA	No	Yes	N.A.
Jubatus	Yes	Yes	No

SAMOA addresses the aforementioned limitations of existing frameworks and tools by:

- SAMOA is a framework that executes on top of distributed streaming computation platforms, such as Storm and S4. Hence, SAMOA inherits the scalability of the underlying platform.

- SAMOA utilizes stream ML paradigm and it contains collection of streaming ML algorithms. Hence it is suitable to perform real-time analytics.

- SAMOA decouple the ML algorithms with the underlying distributed streaming computation platforms, which means we can easily deploy SAMOA in different platforms. In other words, SAMOA has "write once, deploy everywhere" paradigm, i.e. we only need to write the ML algorithms once using SAMOA API and we can easily deploy the algorithms in different types of cluster.

To conclude, SAMOA scales horizontally, is designed for streaming ML paradigm (SAMOA focuses on speed/real-time analytics), and is loosely coupled with its underlying distributed computation platform.

2.3.3 Samoa's Design Goals

- **Flexibility** in term of developing new ML algorithms or reusing existing ML algorithms from existing ML frameworks.

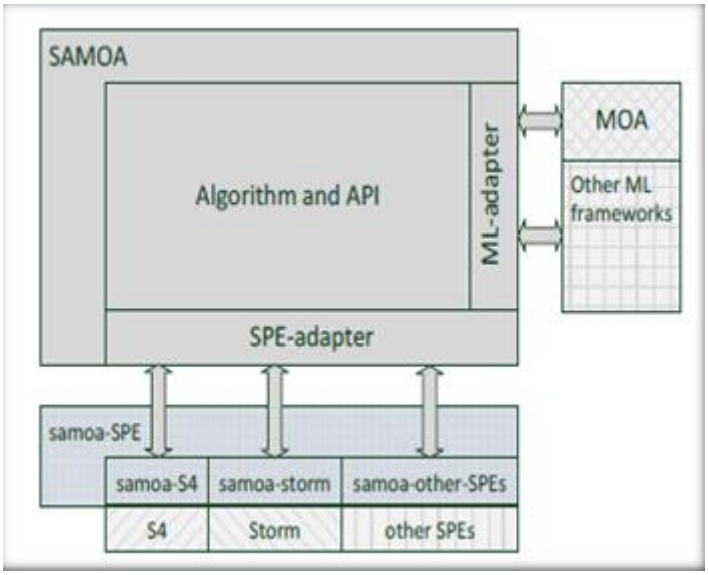
- **Extensibility** in term of extending SAMOA with new SPEs.

- **Scalability** in term of handling ever increasing amount of data.

2.3.4 Algorithms used in SAMOA

- Decision Tree Induction
- Hoeffding Tree Induction
- Vertical Hoeffding Tree Induction
- K-means Clustering
- Gradient Boosted Tree
- Bayesian Model Averaging

Figure 2.2: Samoa's Inner Architecture



2.4 Stream Join Processing Implementations

Stream join processing is a fundamental operation, in which we merge information from different streams inputs. This is especially useful in many applications such as sensor networks in which the streams arriving from different sources may need to be related with one another. In the stream setting input tuples arrive constantly and result tuples need to be produced continuously as well. Join algorithms that use blocking operations, e.g., sorting, are no longer working properly. Conventional methods for cost estimation and query optimization are also inappropriate, because they assume finite inputs. Moreover, the long-running and evolving nature of stream queries calls for even more adaptive processing strategies , such as data structures that help our implementation achieve lower latency and higher throughput, that can react to changes of data and stream characteristics have.

By nature of stream joins, we need to take in consideration the past tuples of the stream, so that adds another dimension to the challenge. In general, in order to compute the complete result of a stream join, we need to retain all past arrivals as part of the processing state, be-cause a new tuple may join with an arbitrarily old tuple

arrived in the past. This problem is exacerbated by unbounded input streams, limited processing resources, and high performance requirements (low latency –high throughput), as it is impossible in the long run to keep all past history in fast memory. Hence, motivated by the inherently high computational complexity of stream joins, a considerable research effort has been devoted to their parallelization. So, much work have been done to create FPGA architectures of the join processing implementations, with a view to achieve speedup to those algorithms.
Hence, in this section we present software and hardware based implementations of Handshake join and Cell join implementations.

2.4.1 Software based implementations

In this section, we present to you the state of the art algorithms, Handshake join and Cell join that are compared with the ScaleJoin algorithm, in terms of processing latency and processing throughput.
Kang et al. [4] were the first to describe a streaming join operator. Each newly arriving tuple r of stream R is processed in three steps: First, scan the window associated with input stream S and look for matching tuples, second Invalidate old tuples in both windows and third, insert r into the window associated with R. Kang's procedure latency characteristics, because it tries to find matches on any arrival tuple.
Gedik et al. [10] have thus suggested to stick with the three step procedure even in parallel environments, but parallelize the scan task over available processing units. Hence, Celljoin inherited the latency characteristics of Kang's procedure and thus, low scalability.
Then, Handshake Join [7], replaced the three step procedure by a data flow oriented setup. Both input streams notionally flow through the stream processing engine, in opposing directions. The two sliding windows are laid out side by side and predicate evaluations are performed along the windows whenever two tuples encounter each other. In this implementation, latency is scaling linearly depending on the time window size, that it is considerable in large windows.
Later, Low Latency Handshake Join was presented [8] and its goal was to remove the latency bottleneck of handshake join. The algorithm is semantically equivalent to the handshake join and classical stream join operators with respect to their set of output tuples. It also preserves the architecture awareness of the handshake join but in addition lowers significantly the latency.

2.4.2 Hardware based implementations

In this chapter, we present to you all related works that have been materialized upon hardware.
To begin with, Halstead *et. al*. [5] introduces an FPGA-based implementation that uses a hash-join engine, achieving impressive performance speedup compared to the corresponding software implementation. The main difference of this work vs. our proposed solution is that this work focuses on relational data bases and not on stream processing.
Moreover, Qian *et al*. presented M3Join in [13]. The implementation of M3Join in hardware was compared vs. software solution, achieving much higher processing throughput.

Then, authors in [7] and [8] presented the best ways to implement Handshake join algorithm on reconfigurable logic by setting up an adaptive merging network. The results in [8] show the evaluation between the proposed model and the baseline and nested loop joins and explains why the proposed model is the best Handshake join operator compared to other hardware-based solutions. In addition, Oge et al. [11, 12] presented a hardware solution for the Handshake algorithm that can process input of 1.2 million tuples per second for small window sizes equal to 512 tuples and 1024 tuples, though. In addition, [14] proposes a scalable and order-agnostic hardware design of sliding-window aggregation and its implementation on an FPGA. The proposed design adopts a two-step aggregation method using panes and supports disordered data arrival with punctuations.

Regarding the Cell join, presented by Bu˘gra Gedik, Rajesh R. Bordawekar and Philip S. Yu [10], an implementation has been made using Cell Processors, that is initially intended for game consoles and multimedia rich consumer devices, for stream join processing. In their work, the implementation's problems are discussed and analyzed in order to find a proper architecture for the SPEs and Cells , discussing in advance the coordinator-side and co-processor-side processing logic of the join algorithm and the optimal basic window size for this operation.

Chapter 3: Algorithm Analysis

3.1: Introduction

In this Chapter, we present an analytical description of the ScaleJoin algorithm. Stream joins are among the most important and expensive data stream mining operators employed to process live data in a real-time fashion. Directly from their database counterpart, stream joins compare tuples coming from data streams rather than relations.

So, the chapter is divided in 6 sections. In the first section we define some terms used in ScaleJoin algorithm, in order to make understandable how the individual parts of the algorithm are functioning. Moreover, in the second section, an introduction is being made about the goals, problems and description of the algorithm and its functions, in parallelization and its concurrent abstract data type. The third Section presents a more analytical description of the algorithm, as it comes to modules separately. In the fourth section, we make a short reference to the algorithms that ScaleJoin is compared, in terms of processing throughput and algorithmic latency. In fifth section, we analyze each separate part of the algorithm, with a view to analyze our thoughts about each one and make the whole implementation simpler on reconfigurable logic. Finally, sixth section is concluding all above information and makes an introduction on the whole architecture that we are going to use.

3.2: Algorithm's terminology

In this section, we describe how the ScaleJoin algorithm defines its basic functions, in order to work properly. In the ScaleJoin report [6], these terms are used to describe the inner functions and their uses in the algorithm.

3.2.1 Ready tuple

A ready tuple is a tuple that it is waiting to be sent to the PUs (Processing Units) by the ScaleGate distributed Data Structure. When sent to a PU, a tuple is compared with all tuples of the opposite stream, which have less or equal than window size timestamp range.

A ready tuple, as defined in [6], is a tuple that it's timestamp ts is less or equal than merge.ts, where merge.ts is the minimum among the latest timestamps from each timestamp-sorted stream j (merge.ts = $\min_j(\text{maxi}(t^j_i .ts))$.

3.2.2 Tuple and Time based Windows

A window is an interval that a function can take place. Windows are described by window size constant which defines the length of the interval. Windows, in stream data mining, can be either time based or tuple based. Time based windows have a defined window size counted in t seconds and the interval defined for a certain tuple, with a timestamp t_k, is $[t_k-t,t_k+t]$, when $t_k-t>0$ and $[0,t_k+t]$, when $t_k-t<=0$. Tuple based windows have a window size counted in k tuples and the interval defined for a certain tuple, which is the m-th tuple, is [m-k, m+k], when m-k >0 and $[0,t_k+t]$, when m-k <=0.

In ScaleJoin report is focused on time-based windows, but with some changes it can easily work for tuple-based windows as well.

3.2.3 Tuple Merge

Merging tuples is one of the basic functions of the ScaleJoin algorithm. As described in the report, PUs are comparing each tuple in each stream with each tuple of the opposite stream to find correlations between tuples. If found, PU creates a merge tuple by combining the two tuples that correlation was found in a tuple t0 with its timestamp calculated as t0.ts = max (t_R.ts, tS.ts), where t_R.ts and t_S.ts are the tuples from stream R and S, respectively, that the correlation was found. Then, the merged tuple is sent to the output ScaleGate, where it be defined as Ready and be extracted as an output of the algorithm.

3.3: The ScaleJoin algorithm

As discussed earlier, ScaleJoin is a Stream Join algorithm, which is parallel and works upon ScaleGate concurrent data structure. ScaleGate is acting 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 them.

ScaleJoin allows for the parallel execution of an arbitrary number of sequential stream joins while distributing the overall work among them without assuming any centralized coordinator. As the evaluation shows in the report, ScaleJoin does not only enforce deterministic processing while providing disjoint and skew-resilient parallelism, but also achieves higher processing throughput and lower processing latency in comparison with the state of the art parallel stream join operators such as Celljoin and Handshake join.

3.3.1 Problems and challenges

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

Deterministic processing means that 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.

Disjoint parallelism, requiring a method not to rely on any centralized coordinator and thus leverage the wide-spreading architectures supporting high degrees of parallelism and the continuously increasing available computational resources.

Skew-resilience, to be able to confront any varying rate of nature of data streams and to enable the processing of tuples that, while referring to the same logical stream, might be delivered by arbitrary numbers of distinct physical streams.

3.3.2 Parallelization

As it comes to parallelization, ScaleJoin algorithm parallelizes its comparisons by using PUs. State of the art parallelization techniques still rely on data types that are oblivious to the specific needs of parallel stream joins. Based on this observation, the possibilities enabled by lifting the parallelization challenges to the articulation points that maintain the tuples consumed and produced by a parallel stream join are being explored. More specifically stream joins are extremely computationally expensive nature, so it is understandable that parallelization improves significally the time that the output needs to be produced. ScaleJoin is using n PUs to run the comparisons in parallel, approximately 1/n of the overall comparisons by each PU. Each PU gets a ready tuple and looks for correlations among the tuples of the opposite stream, that are in the interval of a time window of the processed tuple timestamp, in order to produce output tuples by the correlating tuples. This new parallelization perspective does not only allow to address such challenges and overcome state of the art parallelization approaches but also allows to achieve high processing throughput and low processing latency, as shown in the evaluation of the report[6] and, also, in section 3.4.

3.3.3 Abstract Data Type: ScaleGate

ScaleGate, as mentioned before, is a new concurrent abstract data type acting 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 them. ScaleJoin is built upon ScaleGate, which guarantees properties essential for the parallelization of ScaleJoin. ScaleGate, is a lock-free implementations ensure 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. Also, as proved in the report, ScaleGate algorithm is a linearizable implementation. In other words, it guarantees that every method call

appears to take effect at some point (linearization point) between its invocation and response and more formally, for a linearizable implementation of a data structure.

ScaleGate allows for an arbitrary number of timestamp-sorted streams each delivered by a source entity, to be merged into a timestamp-sorted stream of ready tuples. At the same time, it allows for an arbitrary number of reader entities to consume all the ready tuples of the resulting timestamp-sorted stream. Furthermore, ScaleGate distributes the overall work in the processing units by delivering ready tuples in a round robin fashion, in order to be totally parallelized and have the maximum throughput achieved.

3.3.4 ScaleJoin Basic Steps

Below, we present ScaleJoin basic steps, according to the report. In the fourth section we will analyze how the algorithm is functioning inside.

Table 3.1: ScaleJoin basic Steps

- Merge the input of each stream in a timestamp-sorted stream.
- Send timestamp-sorted ready tuples to PUs, in a round robin fashion.
- Compare tuples with all tuples of the opposite stream, which are inside a window size interval.
- Find matches and create merged tuples.
- Sort by timestamp the merged tuples.
- Output the ready produced tuples.

3.4: ScaleJoin evaluation

As it comes to the evaluation, ScaleJoin is compared with the state of the art parallel stream joins, Celljoin and the Handshake join algorithms. Scalejoin's report mentions the evaluation data set of the algorithm is the same benchmark used to evaluate Celljoin and Handshake join. R tuples are composed by attributes [ts, x, y, z], where x, y, z are of types int, float and char[20], respectively. S tuples are composed by attributes [ts, a, b, c, d], where a, b, c, d are of types int, float, double and bool, respectively. An output tuple composed by attributes [ts, x, y, z, a, b, c, d] is created when |x-a|<10 AND |y-b|<10. The values of the attributes are drawn from a uniform distribution in the interval of [1-10000]. The outcome of the evaluation for all pre mentioned, on an eight-core computer and a window of 15 minutes, are shown in the table 3.1 bellow.

Table 3.2 ScaleJoin evaluation table

Algorithm	Throughput (t/s)
ScaleJoin	1300
Handshake Join	1000
Handshake Join (SIMD implementation)	1400
Celljoin	750
Celljoin (SIMD implementation)	1000

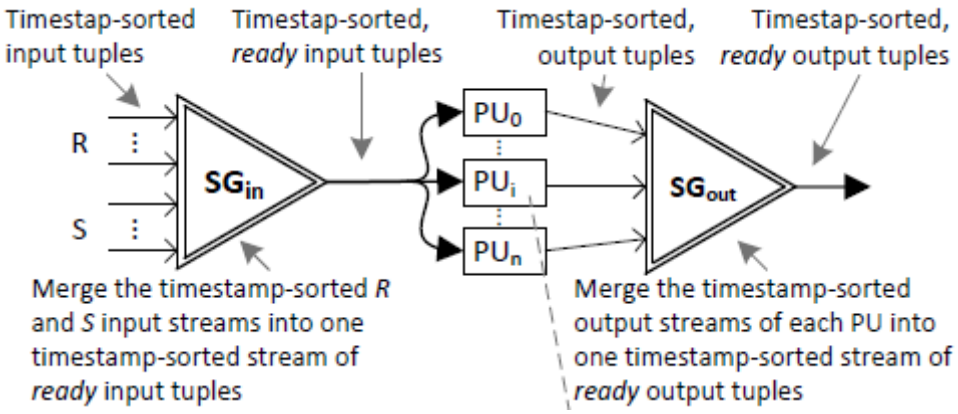
As the table shows ScaleJoin is performing significally better than both Handshake join and Celljoin algorithms when they are not relying in SIMD instructions. When relying on SIMD, ScaleJoin is still performing better than Celljoin but less than 8% worse than Handshake Join. In this point, it should be noticed that only ScaleJoin of these parallelization techniques enforces deterministic processing, and has been evaluated for deployment-specific aspects such as number of physical streams delivering R and S tuples and rate fluctuations.

Moreover, there has been a latency evaluation of those algorithms in the report of ScaleJoin. The original Handshake join algorithm incurs in a processing latency that is proportional to half of the window size (in this data set, the window size is equal to 15 minutes). Though, the improved low-latency Handshake Join algorithm and Celljoin algorithm achieved latencies in the realm of milliseconds, same as ScaleJoin algorithm.

3.5: ScaleJoin analysis

As discussed earlier, ScaleJoin algorithm is a parallel stream join algorithm that compares tuples from each stream, in order to find correlations between them and merge them into a new tuple. In section 3.3, we discussed, briefly, about what the ScaleJoin algorithm does. Therefore, in this section, our major concern is how the algorithm works, how each separate module is operating and how all modules work together. Basically, as shown in the figure 3.2, which is taken by the report [7], the ScaleJoin algorithm has 3 parts. The first one is the ScaleGate (SG), which works as an input of the whole algorithm and called SGin, data type that has as inputs each Streams tuples. The second one is the processing units (PUs) part, which has n PUs and their inputs are the ready tuples given by the SGin. The last part is another ScaleGate data type, which has as an input the output tuples given by PUs and it is called SGout, because it is the output of the whole algorithm.

Figure.3.2: Overview of Scalejoin's architecture



3.5.1 ScaleGate input (SG_{in})

Let 2 streams, R and S. SG_{in}'s input is an arbitrary number of R and S tuples delivered by streams need to be merged into a timestamp-sorted stream of ready (defined in section 3.2) tuples in order to ensure deterministic processing. ScaleGate is merging the input streams' tuples in a stream and defines as ready tuples the tuples that are ready to be processed by the PUs. Incoming tuples are still sorted by timestamp when others are processed, in order to be compared with all the tuples of the opposite stream and have the appropriate result as merging tuples. Specifically, the above data structure is used for merging the incoming tuples from various number of streams into a single timestamp-sorted stream without using any locks. In addition, the ScaleGate module distributes the incoming tuples to the parallel processing threads without the need for centralized coordination. Last, the ScaleGate module is, also, used for collecting the correlated output tuples from the parallel threads to the final output.

ScaleGate has 2 major functions, the **addTuple** function and **getNextReadyTuple** function. AddTuple allows a tuple from the source entity to be merged by ScaleGate in the resulting timestamp-sorted stream of ready tuples, concurrently with other operations. GetNextReadyTuple which provides to the calling reader entity the next earliest ready tuple that has not been yet consumed by the former, concurrently with other operations.

During the insertion of a tuple, the appropriate position is located by starting the search from the highest level node that is closest to the last added tuple from the same source, since the tuples from each source arrive in increasing timestamp order. The main advantage of this software-based module is that it offers lock-free and high-performance processing of streaming data without taking into account the number of the incoming or the outgoing streams.

3.5.2 Processing units

As mentioned previously in this thesis, Processing units (PUs) are the major parallelization of the ScaleJoin algorithm. PUs split equally, approximately 1/n, the overall comparisons that the ScaleJoin algorithm has to do, that can be calculated by knowing an average incoming tuple rate (tuples/sec) and the window size parameter, always referring to time based windows, and is 2 x WinSize x tupleRate^2 comparisons. As it is shown in figure 3.2, PUs are processing ready tuples, explained in section 3.5.1, given by the SGin module.

As mentioned in the report, the work done by a Processing unit is almost similar with Kang's three step procedure [3], in which we have 3 tasks. Three step procedure tasks are, first to. Compare t_R with all t_S W_S, second add t_R to WR and third to remove all t_i in W_R: $t_i.ts < t_R.ts$ – winsize. This procedure differs from the ScaleJoin algorithm in the way the tuples are stored in their respective windows. In this new procedure, R and S tuples are stored in PU's windows in a round robin fashion. More concretely, each PUi maintains a counter of the ready tuples being processed and stores a new incoming ready tuple only if counter%n is equal to i, in which n is the number of PUs implemented in the algorithm and counter is a simple tuple counter of both streams.

A Processing unit's major part is the combination of 2 tuples. Whenever a correlation is found, PU has to combine a tuple, like explained in section 3.2, create a new tuple and add it to a secondary output ScaleGate as shown in image 3.2. The correlations are checked, as referred in the report, using a predicate and the function **holds**, which checks if 2 tuples are correlated. Each comparison that has to be done must be done only by one PU, in order to have the right results.

Figure 3.4 shows the flowchart of a PU, based on pseudocode given in the report.

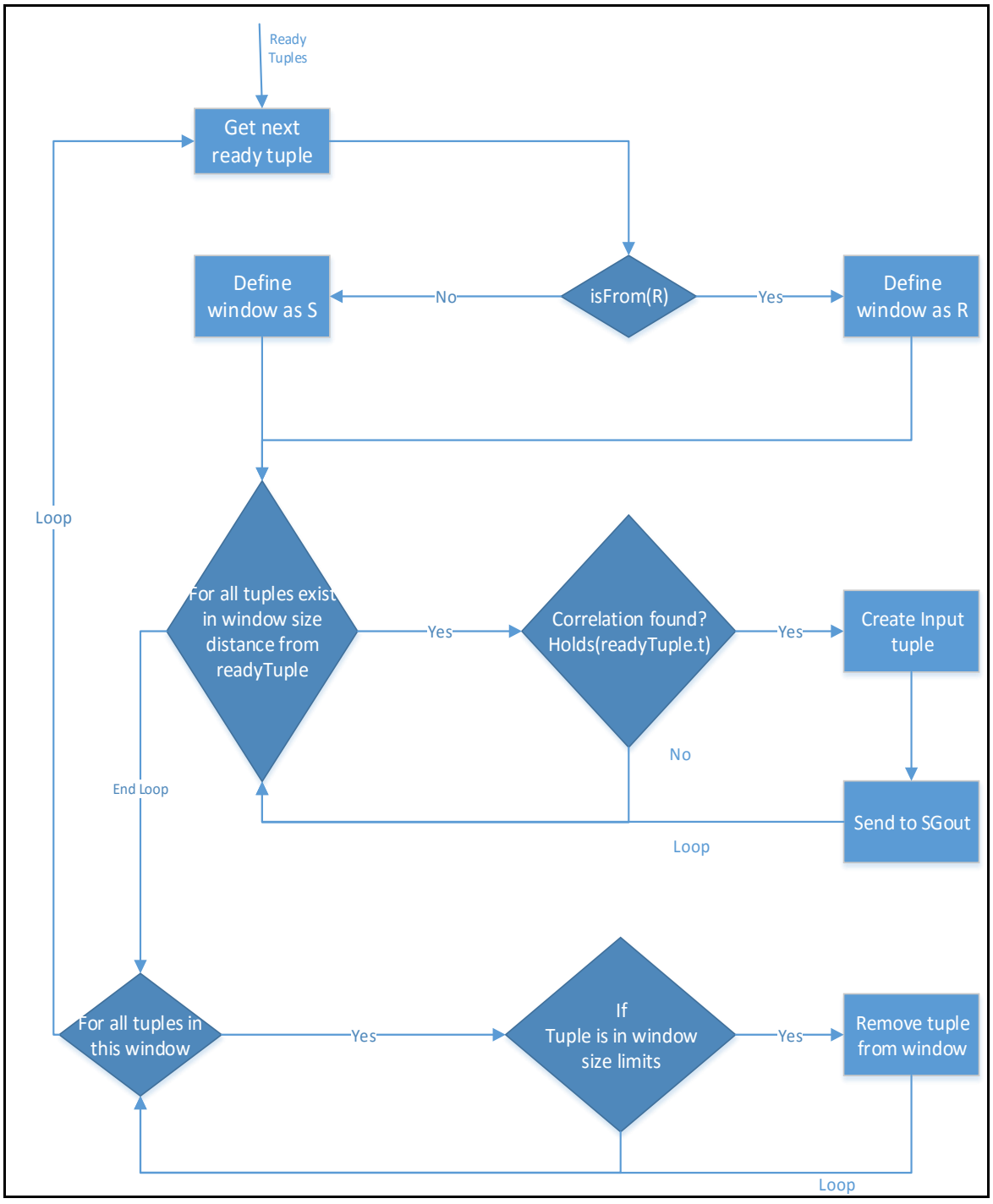
3.5.3 ScaleGate output (SG_{out})

Scalejoin's output is handled by another ScaleGate. ScaleGate is fed by merged tuples, explained in section 3.2, in order to be timestamp-sorted and be produced as an output of the whole algorithm. As mentioned before, the output tuples must be ready to be given as an output.

3.5.4 Conclusion

To sum up this section, as shown in figure 3.2 the ScaleJoin algorithm uses 2 ScaleGate implementations and N Pus in its software implementation. ScaleGate implementations are used only as sorted timestamp-based storage to keep the tuples coming from S and R or the output. The processing is done by the Pus, which compare each tuple from both stream and split the whole process depending on the number of Pus mapped, i.e. the number of processors in the system. ScaleGate are managing the I/O in parallel with the Processing Units.

Figure 3.3: PU flowchart



3.6 Thoughts and processing steps

In this section, we analyze the processing steps of our implementation, based in what we saw in this chapter about the ScaleJoin implementation. We start by showing our implementations steps, how the comparisons are made and how we ensure deterministic processing through our implementation and finally how we output the appropriate results.

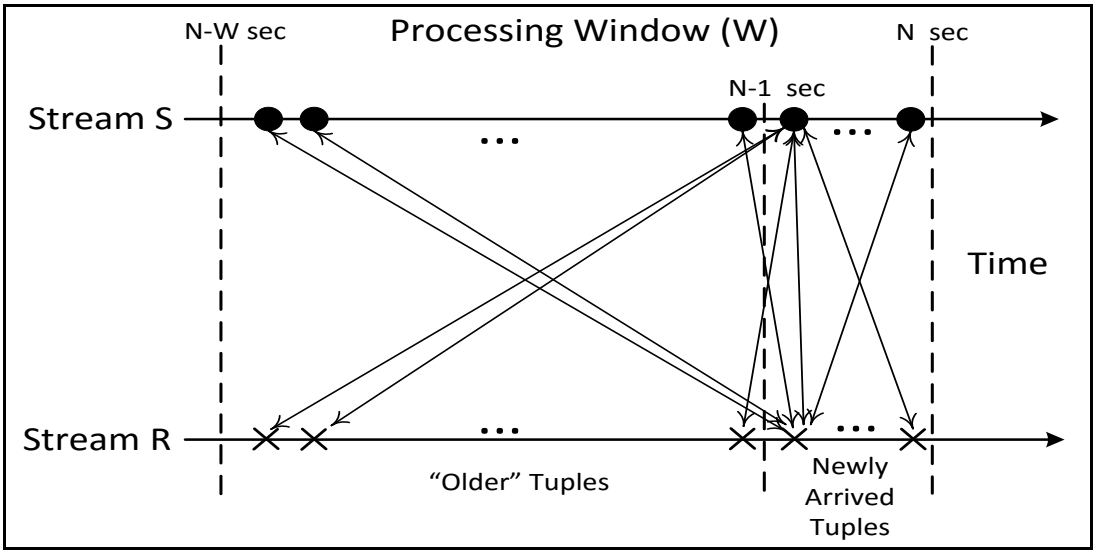
At first, we make a reference to the algorithm steps, which we need to implement in order to work correctly.

1. Load all the new tuples.
2. Make all the comparisons for each one of those tuples.
3. Store and output the merged tuples, timestamp sorted
4. Delete tuples that have made all the comparisons.
5. Start again.

So, in order to make an accurate implementation we must take in consideration those parts of the algorithm. Moreover, we need to think about the nature of stream and be prepared for each rate of incoming tuples. So, given a window size and the incoming tuple rate, our comparisons can be shown in the figure below. More specifically, every second we need to compare S stream's new tuples with all prior ones from R, which timestamp is greater or equal the difference of the current timestamp and window size, respectively for R stream's new tuples with all S stream's tuples prior ones. Additionally, we have to compare each R and S new tuples but only once (!), in order not to have duplicate tuples.

By definition, each tuple in stream joins need to be compared with all tuples within the interval [N-W, N+W]. In figure 3.4, tuples from the N-th second are only compared with the prior tuples i.e. [N-W, N]. Thus, within the next seconds, each one of the arriving tuples, will also be compared with the tuples of the N-th second. So, after Window size seconds all comparisons have been done with the tuples from the N-th second, and finally, those tuples will be deleted (overwrote) and the process will continue.

Figure 3.4 Correlation Graph



3.7 Conclusion

ScaleJoin algorithm is a stream join algorithm and therefore it has high complexity. By knowing the full functionality of the algorithm, its challenges and the total process needed, we can implement it in reconfigurable logic. As mentioned before, we to fully understand the nature of a stream and make the most of the incoming data, achieving, at the same time, deterministic processing, disjoint parallelism and skew-resilience. Additionally, understanding each successive module as a distinct entity is useful enough, in order to successful map it in reconfigurable logic. Then, by taking into account the algorithm's processing steps we can accomplish the correct result and have a successful implementation.

Chapter 4: ScaleJoin Architecture

4.1 Introduction

In this chapter, we present the final architecture of the ScaleJoin algorithm in reconfigurable logic. The final architecture cooperates with a coprocessor for the implementation of ScaleJoin. The architecture is mapped on Convey platform using four Virtex 6 LX760 FPGA devices. By using Convey platform, we can accurately fulfil Scalejoin's challenges and successfully simulate a stream's behavior. Moreover, the whole implementation's input is controlled via a C code, which stores each tuple of both streams in Convey's RAM. Further explanation will occur later in this chapter.

This chapter is divided in 5 sections. The first one refers to the Convey HC-2 architecture and its tools. The second one is focusing in the understanding of the implementation's top level and its sub modules, the third focuses on each ScaleJoin module mapped in the implementation, the fourth focusses on the parallelization of the processing units, which are mentioned before. Lastly, we explain the structure of a Processing Unit in section 4.

4.2 Convey HC-2

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.

4.2.1 HC-2 server

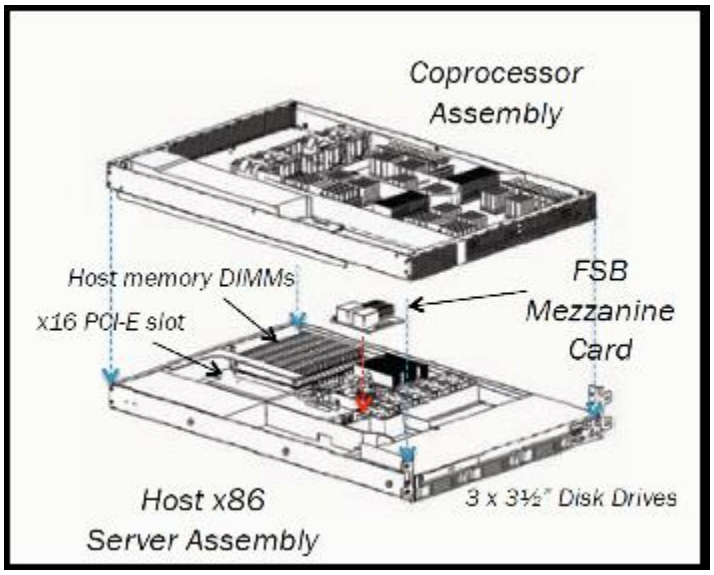
Convey's infrastructure combines an Intel Xeon processor and a Convey designed coprocessor based on Xilinx Field Programmable Gate Arrays, with its own high-bandwidth, virtual memory addressed, cache coherent memory subsystem. It also offers an ANSI standard development environment, increasing productivity and portability. HC-2's main strength is Convey's implementations, called Personalities, which are extensions to the x86 instruction set that are implemented in hardware increasing productivity and optimizing performance of specific portions of an application. They are sharing the same physical and virtual address spaces with the x86 instructions, and applications can contain both x86 and coprocessor instructions in a single-instruction stream. Convey compilers generate one executable image that

contains both x86 and coprocessor instructions. Systems can contain multiple personalities. Convey provides a Personality Development Kit (PDK) for creation of new application oriented architectures discussed in details later in this chapter. Another strong point of HC-2 is its memory, which provides a bandwidth of 80 Gigabytes/sec delivering huge sustainable performance. A shared virtual and physical memory between the coprocessor and the x86 provide the tight integration that allows the system to be programmed as a single architecture. This means that the programmer does not need to manage the physical memory on the coprocessor nor explicitly move data back and forth between the x86 main memory and the coprocessor main memory.

4.2.2 Hc-1 system architecture

Convey HC-2 is a hybrid-core computer system that uses a commodity two-socket motherboard to combine a reconfigurable, FPGA-based coprocessor with an industry standard Intel 64 host processor. Physically, the system is based on two main logic boards in a rack-mountable 2U enclosure. The top half of the enclosure is the coprocessor and the bottom half is the commodity motherboard. A mezzanine interconnection mechanism connects the halves and extends the host motherboard's front-side bus (FSB) to the coprocessor. The entire system consumes approximately 600 watts with the coprocessor executing code. The system architecture is shown in Figures 4.1 and 4.2 below.

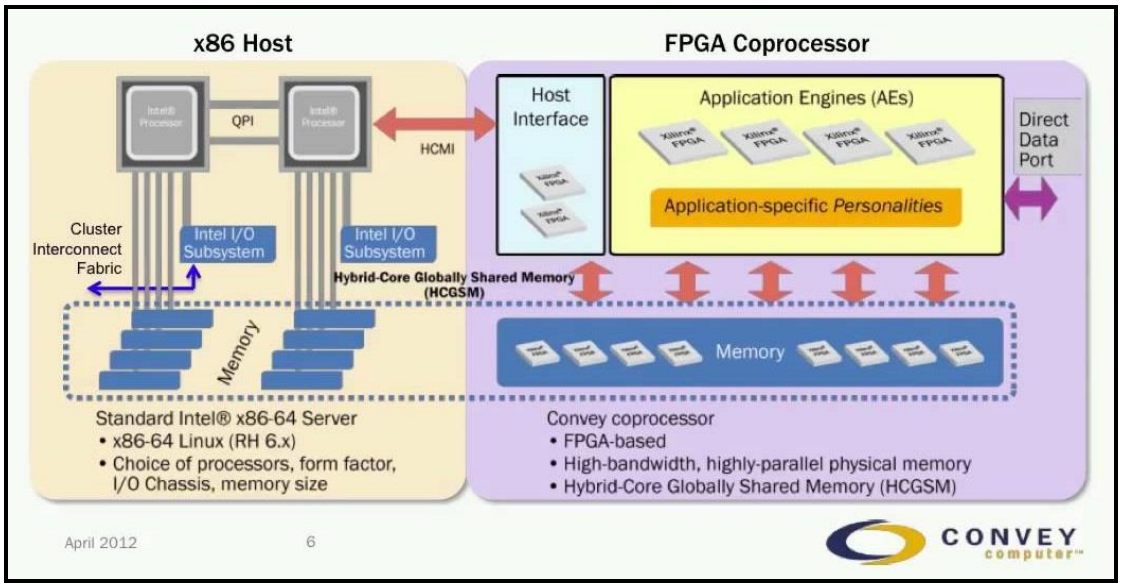
Figure 4.1 HC-1 server architecture.



4.2.3 Intel host processor

HC-1's host consists of a dual socket Intel server motherboard an Intel 5400 memory-controller hub chipset, 1,066 MHz FSB and a 2.13 GHz dual core Intel Xeon low voltage processor. The HC-2 host runs a 64-bit 2.6.18 Linux kernel with a Convey modified virtual memory system for memory coherent with the coprocessor board memory reasons.

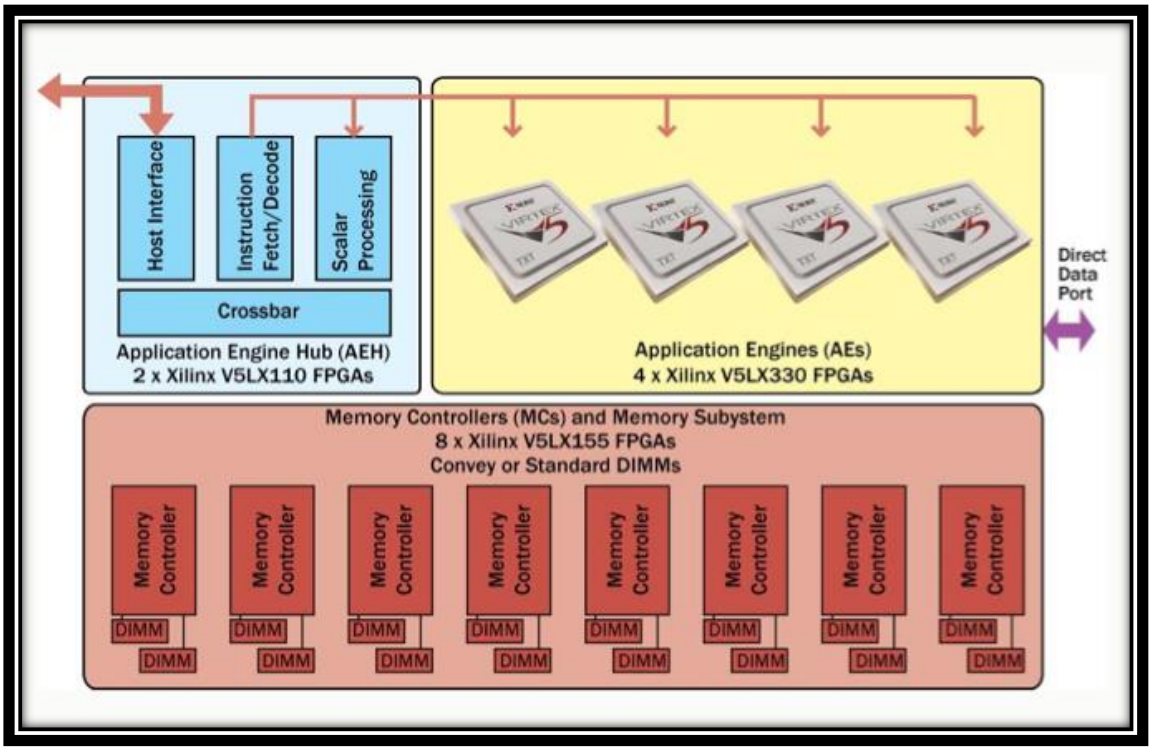
Figure 4.2 HC-1 Architecture.



4.2.4 Convey FPGA-based coprocessor

Convey's HC-1 coprocessor composed by three main sets of components. The Application Engines (AEs), the Memory Controllers (MCs), and the Application Engine Hub (AEH).

Figure 4.3 Coprocessor three main components AEH, AEs, MCs.



The Application Engine Hub (AEH) is the coprocessor's to host interface, it consists of two non–user programmable Xilinx V5LX110 FPGAs. One serves as the physical

interface between the coprocessor board and the FSB, it monitors the FSB to maintain the snoopy memory coherence protocol and manages the coprocessor memory's page table. This FPGA is actually mounted to the mezzanine connector. The second one contains a soft-core scalar processor, which implements the base Convey instruction set. It is also the mechanism by which the host invokes computations on the AEs. To support the bandwidth demands of the coprocessor, 8 Memory Controllers (MCs) are used. Each memory controller is implemented on its own FPGA and is connected to two standard DDR2 dual inline memory modules (DIMMs) or to two Convey-designed scatter-gather dual inline memory modules (SG-DIMMs), containing 64 banks each and an integrated Stratix-2 FPGA. The SG-DIMMs allow access to physical memory by quad words (8 bytes) instead of by 64-byte cache lines (as the host does). Accessing by 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 one eighth 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. 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. The way that they process the instructions depends on the personality. The AEs are interconnected with 668 Mbytes/s, full duplex links for AE to AE communication. Each AE has a 2.5 GB/s link to each memory controller, and each SGDIMM has a 5 GB/s link to its corresponding memory controller. The effective memory bandwidth of the AEs is dependent on their memory access pattern to the eight memory controllers and their two SG-DIMMs. Each AE can achieve a theoretical peak bandwidth of 20 Gbyte/s when striding across eight different memory controllers, but this bandwidth would drop if two other AEs attempt to read from the same set of SG-DIMMs because this would saturate the 5 Gbytes/s DIMM memory controller links [12]. The Convey memory system using Scatter/Gather DIMMs has 1024 memory banks. The banks are spread across eight memory controllers (MCs). Each memory controller has two 64-bit busses, and each bus in accessed as eight sub busses (8-bits per sub bus). Finally, each sub bus has eight banks. The 1024 banks is the product of 8 MCs * 2 DIMMs/MC * 8 sub bus/DIMM * 8 bank/sub bus.

4.2.5 Programming model

In Convey's programming model applications can be coded in standard C, C++, or Fortran, the AEs act as co-processors to the scalar processor, while the scalar processor acts as a co-processor to the host CPU. Because of this, the executable file on the host contains integrated scalar processor code. This is transferred to and executed on the scalar processor when the host code calls a scalar processor routine through one of Convey's runtime library. The scalar processor code can contain instructions that are dispatched and executed on the AEs. The final executable is generated by a unified compiler and it integrates both x86 and co-processor

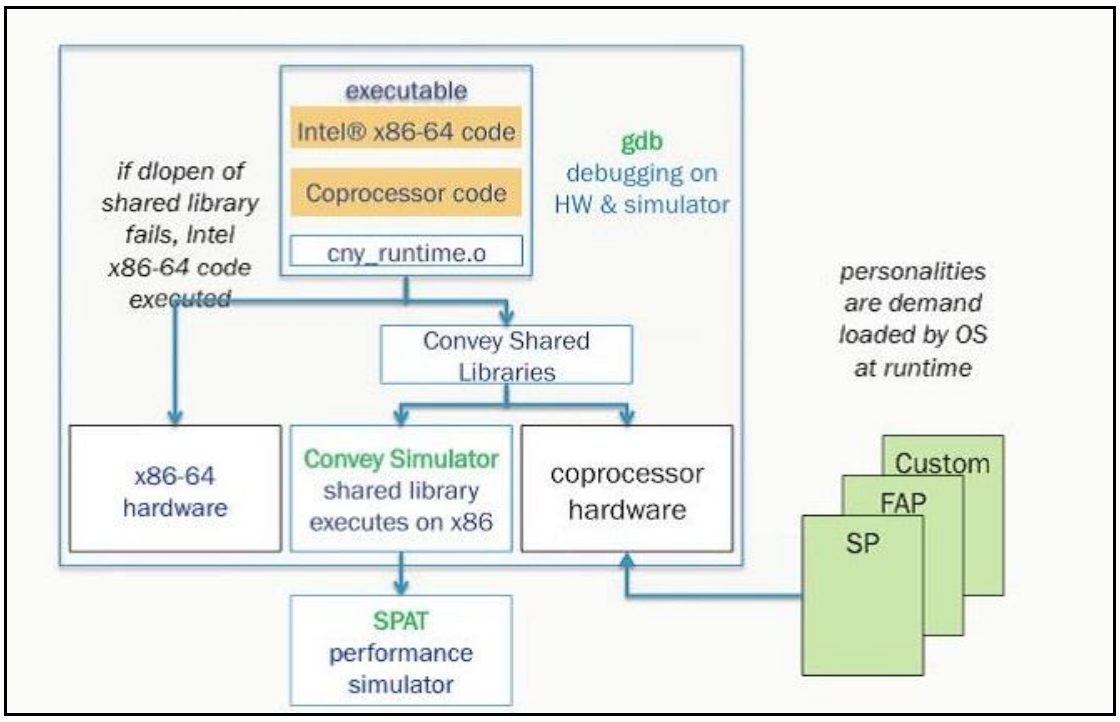
instructions defined by the personality used on compilation time. Figure 4.4 shows an abstract view of the programming.

4.2.6 Porting already existing applications

Convey offers four different ways for porting already existing applications to their HC-2 system mentioned below:

- Use the Convey Mathematical Libraries (CML), which is a set of functions optimized for the co-processor, which use predefined Convey-supplied personalities, for example Convey's CML-based FFT uses the single-precision personality.
- Compile one or more routines with Convey's compiler. This uses the Convey auto-vectorization tool to automatically select and vectorize do/for loops for execution on the co-processor. Directives and pragmas can also be manually inserted in the source code, to explicitly indicate which part of a routine should execute on the co-processor.
- Develop hand-code routines in assembly language, using both standard instructions and personality specific accelerator instructions. Which can be called from C, C++, or FORTRAN code.
- Develop a custom personality using Convey's Personality Development Kit (PDK), to give the ultimate in performance using a hardware description language such as Verilog or VHDL.

Figure 4.4 HC-2 runtime environment



4.2.7 The Convey Personality Development Kit

The Personality Development Kit is a set of tools and infrastructure that enables development of a custom personality for the Convey HC-1 system. A set of generic instructions and defined machine state in the Convey instruction-set architecture allows the user to define the behavior of the personality. Logic libraries included in the PDK provide the interfaces to the scalar processor, memory controllers, to the inter-FPGA links and to the management processor for debug. We will present each of these interfaces with more detail in next session. The user develops custom logic that connects to these interfaces.
The Convey PDK provides the following set of features as a part of the kit:

- Makefiles to support simulation and synthesis design flows.
- A Programming-Language Interface (PLI) to let the host code interface with a behavioral HDL simulator such as Modelsim or Synopsys.
- FPGA hardware interfaces provided as Verilog modules, these interfaces connect custom personality hardware to instruction dispatch, management and memory resources on the coprocessor.
- Custom personality software and hardware simulation environment Bus - functional models are provided to connect each of the hardware interfaces to Convey's architecture simulator.
- A sample personality illustrates how to use the hardware and simulation interfaces to develop a custom personality.

In addition to the PDK package, several Convey software packages are required for PDK development. Which are:

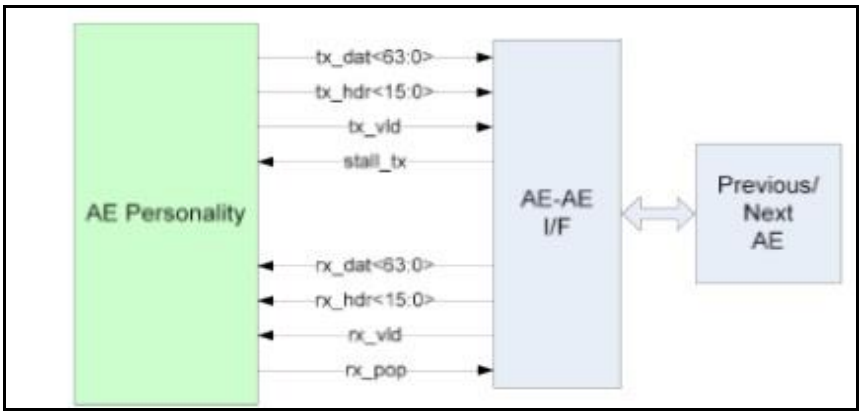
- Xilinx ISE Design Software for synthesis, place and route of FPGAs.
- An HDL simulator for Verilog/VHDL simulation. Mentor ModelSim or Synopsys VCS.

The PDK's simulation framework is easy to use and allows users to switch between a simulated coprocessor mode and an actual coprocessor, by changing a single environment variable.

4.2.8 AE-to-AE Interface

The AE-to-AE interface allows data to be transferred directly from one AE to another. Because the use of an AE-AE interface is unique to each application, it is difficult to design a solution that would be ideal for all custom personalities. Convey provides an AE- AE interface that the user may choose to use. The user is also free to use the signals between AEs in whatever way best supports their application. The Convey provided the AE-to-AE interface, which is designed with unidirectional busses to and from the previous or next AE. Each instance of the interface connects to a single AE, to connect an AE to both the previous and next AEs, two interfaces must be instantiated. This interface is simple and generic so that it can be used by many applications. Figure 4.5 below shows the AE-AE interface to the Custom Personality.

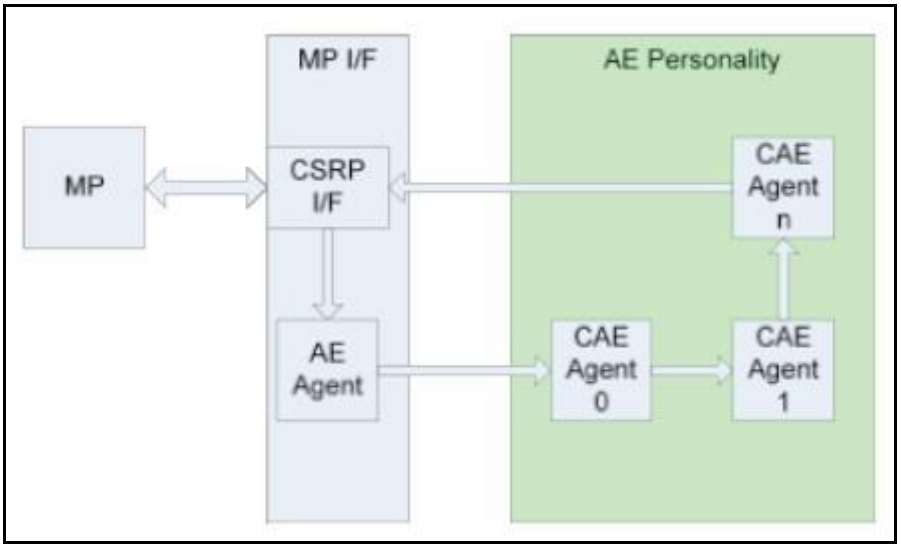
Figure 4.5 AE-to-AE Interface Diagram.



4.2.9 Management/Debug Interface

The management interface provides the communication path between the Management Processor and the AE. The Management Processor (MP) is responsible for initialization and monitoring of the FPGAs. Since this path is independent of the instruction dispatch path from the host processor, it can be useful in debugging by allowing visibility into internal FPGA state, even when the application is hung. The MP interface is instantiated in the Convey-supplied libraries, along with CSR agents in a ring topology. The custom personality must complete the ring by either adding one or more CSR agents to the ring or by simply connecting the inputs to the outputs. For many designs, a single agent is sufficient. For more complicated designs, the developer may choose to instantiate multiple CSR agents. The ring topology allows multiple agents to be placed near their associated logic. PDK CSR Registers are accessed from the host for debugging reasons. The host communicates with the MP FPGA via telnet. Figure 4.6 below shows the connectivity of the CSR interface:

Figure 4.6 Management Interface Diagram.



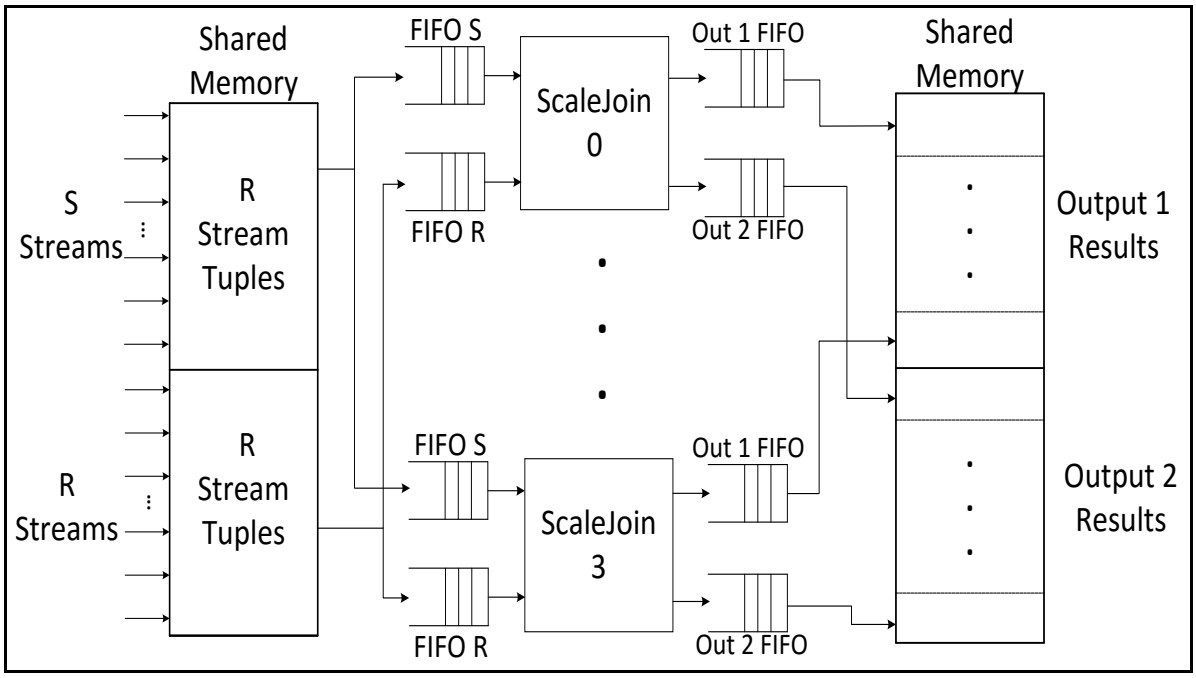
4.3 Reconfigurable ScaleJoin System

This section describes the complete stream join system architecture. This architecture can take advantage of the high scalability and performance advantages that hardware can offer, if it is mapped on a shared memory platform, which can offer fast data I/O data links, and has a large number of available hardware resources. As mentioned before, the proposed architecture was mapped on a Convey HC-2ex FPGA-based server, which permits the communication between the software that is executed on a processor with a reconfigurable subsystem via shared memory. The HC-2ex platform offers an all-around versatile coprocessor memory subsystem that is able to provide nearly-constant streams of data even for non-sequential accesses. There are two main advantages in used HC-2ex platform: first, it offers the ability to build FPGA-based architectures, which are implemented on the Convey reconfigurable coprocessor; second, it offers an aggregate stream bandwidth of up to 80 Gigabytes/sec, which is very useful in cases of data-intensive problems.

Figure 4.7 presents the total system architecture for the stream join processing. As referred above, the proposed architecture is scalable, as it can map up to N Processing Units (PUs), where the number of N is only limited by the available resources. In our prototype platform, each one of the available 4 FPGA devices maps a ScaleJoin module, which has 256 PUs in total, i.e. 128 PUs for each stream. We parallelized the problem by loading different newly arrived tuples into each one of the available ScaleJoin modules from the global shared memory. Hence, our proposed system can process in parallel up to 1024 newly arrived tuples, i.e. 4 x 256 tuples, but it has no restrictions as far as the size of the processing time window.

As referred above, each ScaleJoin module processes a different part of the newly arrived tuples, thus it reads the needed tuples from different parts of the shared memory. The Convey HC-2ex FPGA-based server has 16 parallel memory controllers, which can access concurrently the internal RAM. The newly arrived tuples are stored in shared memory by different threads that serve the physical input streams. Next, the tuples are loaded from the RAM and they are streamed into FIFOs and then they are passed to the processing elements. The combination of the parallel writing in memory at different places and the input FIFOs implement the corresponding software-based ScaleGate data structure, which is presented in [6], for the hardware solution. Also, the PUs are connected in a pipelined way, in order to make all the comparisons needed with the minimum amount of memory reads. Last, each ScaleJoin module outputs the results into an output FIFO and then the results are passed to the global shared memory.

Figure 4.7 FPGA-based Architecture



At this point, we shall make a reference on how the proposed system can offer solution for even higher throughput rates for the incoming streams. The streams usually have a time-based rate of incoming tuples per second. As mentioned above, each FPGA device can process in parallel only a portion of the newly arrived tuples. For our prototype platform, each ScaleJoin module can map up to 256 tuples, i.e. 128 tuples for each stream. Thus, the proposed solution can process up to 1024 newly arrived tuples in parallel. On the other hand, the high level of parallelism that hardware can offer and the high bandwidth data I/O links that our proposed platform offers, leads to the fact that the reconfigurable part can be reloaded with newly arrived tuples at the same rate-based portion of time, i.e. second. This reloading procedure does not reduce the streaming nature of our proposed solution as it takes place on the same portion of time until the new streaming tuples from the next second arrive. This reloading process can take place many times during the same rate-based time portion. Thus, this is the limiting factor as far as the scalability of our system. This process is the main difference, which our proposed solution offers for higher scalability vs. all of the previously proposed hardware based solutions of the stream join processing.

4.2 Scalejoin's Implementation

In order to make the algorithm work correctly, we implemented our design in Convey hybrid Computer ™ HC-2ex. Using this platform, we can have a more realistic simulation of our work and correct various bugs, which would alter the output. To begin with, we used a C code, which generates random tuples for both streams and feeds them into Convey's RAM. Given a Rate size R and a window size W, this C code feeds the algorithm with random tuples and in the same time it sends R and W values

as an input to our implementation. Also, we send the addresses of each array used i.e. 6 arrays for each attribute needed in the process (ts, a , b for stream S and ts, x, y for stream R) and 2 addresses to store the merged output.

Furthermore, the appropriate addresses of the attributes of each stream is been given to the top level, via an Assembly code which just organizes the inputs and the outputs. In order to keep the processing deterministic and valid at the same time, we make use of FIFOs in the input to make sure that the right combination of attributes is being feeded to the ScaleJoin module. Input tuples are sent to the ScaleJoin module, whether an input control decides that they are ready to get fed. It can be also said that the combination of the input FIFOs and their control is a separate architecture of the ScaleGate data type given in the initial report (SG_{in}).

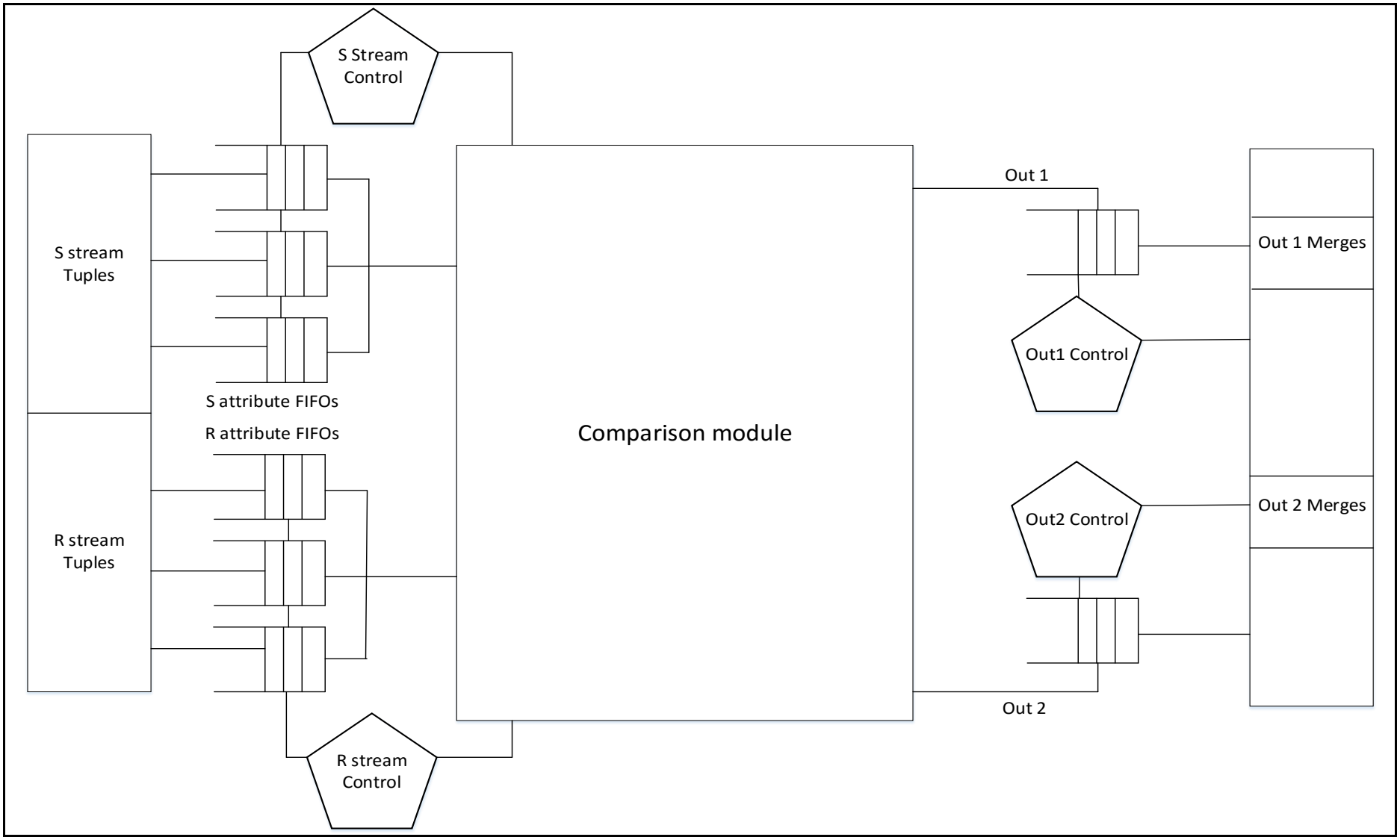
Then, the tuples are fed and loaded in the algorithm's module exactly as it is done by the initial algorithm given. Each time we load the amount of new and ready tuples that came in the last second and then we compare each one of them with the tuples came in a Window size interval before the current second. While we compare each pair of tuples between them, we may have some merges found. So a round robin fashion reading control reads each Processing unit, to find those that have valid outputs and extract them. This phase stops when all tuples needed to get processed are loaded in the ScaleJoin module and all phases of this module are finished. Then, an output signal informs the top level control that this module have finished or it is in the idle phase.

Moving towards the output, all tuples extracted from the ScaleJoin module is being stored in FIFOs, one for each output. Then two control module takes over and gives permission to each one of the FIFOs to write their merged value using the same memory controller for both of them. As said before, this is also an accurate implementation of the ScaleGate data type for the output (SG_{out}). When all tuples from both FIFOs are written in the output addresses, given by the C code, then we ask permission from the memory controller, in order to write the final value to those addresses. This value is the number of merges found in each one of the sets of processing units.

The final step is to inform the C code that this processing phase is finished. So, after each run, we read both arrays which have the output values. From the previous step, during the final write, we write the amount of valid merges found in each array. Thus, with a loop branch, we write all merged outputs into separate .txt files, so if we want to check a specific run and view this runs values, we just open this file and check the merges found. As a summary, when the algorithm makes all the amount of runs in the respective second, it return a sum of the merges found, the window size and rate per second given, the number of runs and the average merges per run found.

Each module will be explained further in this section, previewing a top-down analysis of the implemented algorithm.

Figure 4.8: ScaleJoin architecture



4.3.1 System input (SGin)

As we described in the previous section, a C code creates tuples for both R and S stream and stores them in Convey's RAM. So, our goal at this point is to address them and read them correctly, in order to process them with the implemented algorithm module. As show in graph 3.1, in each second we have to load the new tuples from each stream in the first phase and then compare them with all tuples within a Window size interval, taking into account that we must compare the new tuples came between them only once! To do this, we need two separate but pretty much similar modules.

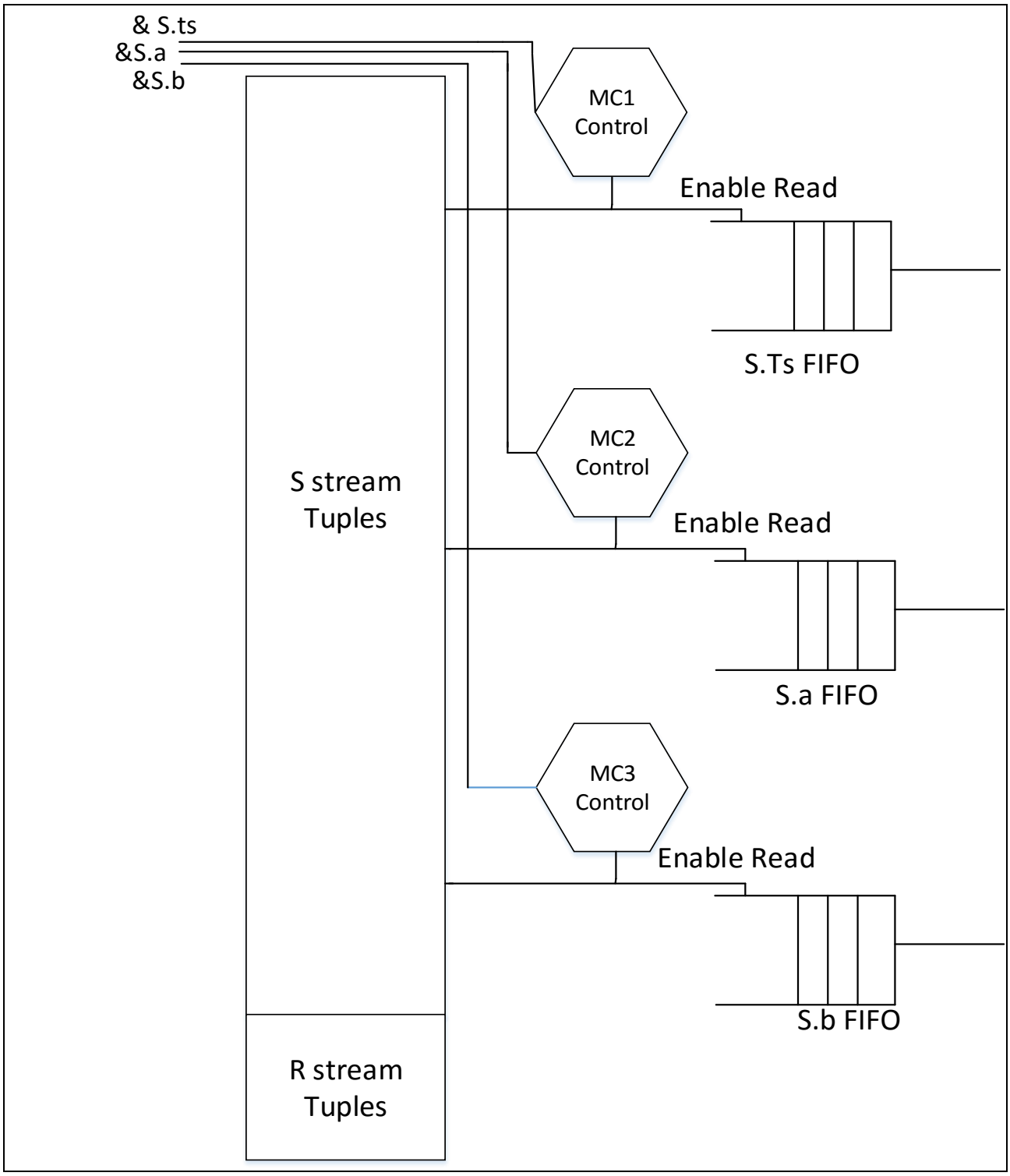
At this point, we have to clarify how the memory inside the Convey platform is organized. In the start of the initial execution, we make use of a malloc-like Convey's instruction which "binds" a specific amount of memory for each attribute needed for both streams. So, for each one of those attributes we make use of a 64-bit value. The size of each array, which contains the attributes, is windows_size*rate. The incoming attributes in those arrays are stored FIFO-like in those arrays, in more details let an array A[n]. In each run, we load all new values in the array, overwriting the tuples inserted window_size seconds before! In this way, we make sure that always the tuples that stay in the reserved arrays are valid for the comparisons of next processing run.

As we mentioned before, when we enable the algorithm to run, we give as an input to the top level the addresses of each attribute of the streams. In the first phase, we read the new tuples, stored in arrays via the C code, in order to store them into the algorithm's FIFOs. When this is done, we need to read all the previously stored tuples, sequentially, with a view to comparing all of them with each one of the new tuples. When this is finished, the one stream has finished its compares but the other one need to compare its new tuples with the others streams new ones. So, the only thing that needs to be done is to read the new tuples of the one stream again and send them to be processed in the algorithm.

In order to give a more analytical example, figure 4.9 shows the full implementation of these specific modules. Regarding S stream's attributes, the addresses of their arrays are sent to the memory controllers control (MC_Ctrl). We use these control modules to read sequentially all the tuples needed in order to have a complete process. When an attribute is read, we increase a counter value, which refers to the current address of each memory controller, by 8 bytes in order to get the next attribute of the array and we enable the Enable read signal to store the value in its respective FIFO. This process continues until we have read all attributes needed and then these modules become Idle and the same process occurs for all attributes in each stream, that are necessary for the whole process.

This module is also shown briefly in the left end in figure 4.8 for both S and R and depicts the whole algorithms input.

Figure 4.9: ScaleGate in Architecture



4.3.2 ScaleJoin module

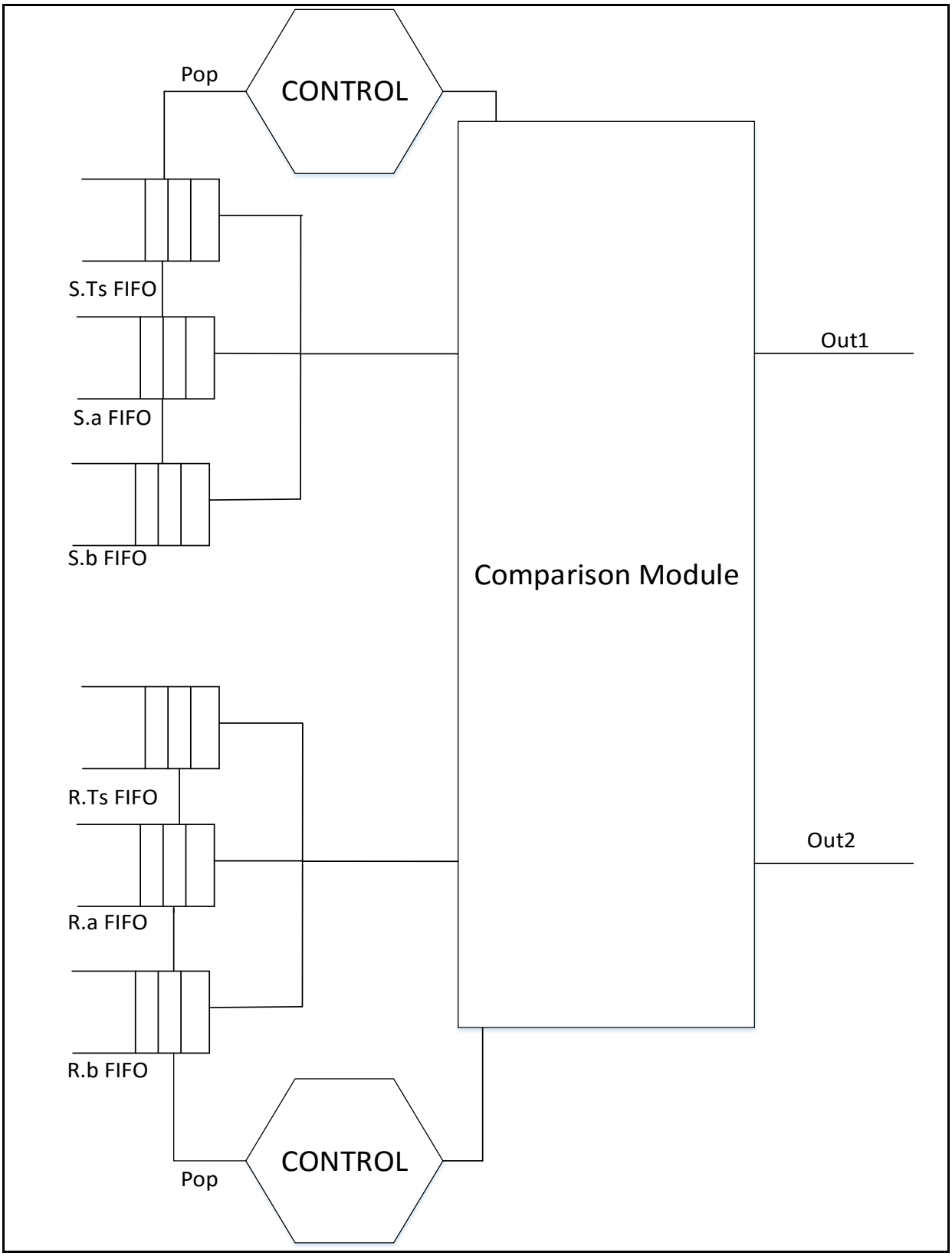
The previous module described is used as an input to the whole implementation. Hence, this section focuses in describing the ScaleJoin module as an entity to the whole algorithm. As we said before, the input of the FIFOs in figure 4.8 are the attributes that are used to make the comparisons needed to complete a process. FIgure 4.9 depicts the next phase of the inserted attributes, which is the processing phase. Further analysis in the architecture of the comparison module will occur in section 4.9.

The output of the memory reading FIFOs are concatenated and inserted to either Stream S or stream R. Control module is informed when all attributes of each stream are ready to feed the tuple in the ScaleJoin module, exactly like the initial ScaleJoin implementation to keep our process deterministic. For example if we insert 3 attributes, 32 bits each, then we insert 96 bits in the algorithm for each tuple in each stream. Moreover, we insert the current rate value and size value to the FPGA via the C code in order to inform this module about the number of tuples it needs to insert in each state of its process. By doing that, we can run any process with varying values of rate or window size in order to conserve the other two challenges mentioned in Scalejoin's report which are skew-resilience, namely, to cope with the bursty and rate-varying nature of data streams and, as we will explain more analytically in section 4.9, Disjoint parallelism, which means not to rely on any centralized coordinator throughout the process.

Regarding to the output of this module, we have two outputs that return the merges between two tuples and we use a large, parallel module for the comparisons of each stream. So having two stream, will result in two outputs that the one is the merges found in the lastly inserted tuples of Stream R and the other output the merges found in the lastly inserted tuples of Stream S. The valid output signals are nothing more than 1 bit values that inform the next module that this outcome is valid one and it needs to be stored. Figure 4.10 depicts the architecture of the comparison module, its inputs and outputs

A more analytical approach about this module will occur in section 4.4, in which we will focus on describing this module's top level and make a top down presentation of each module used to make a successful implementation.

Figure 4.10: Comparison module

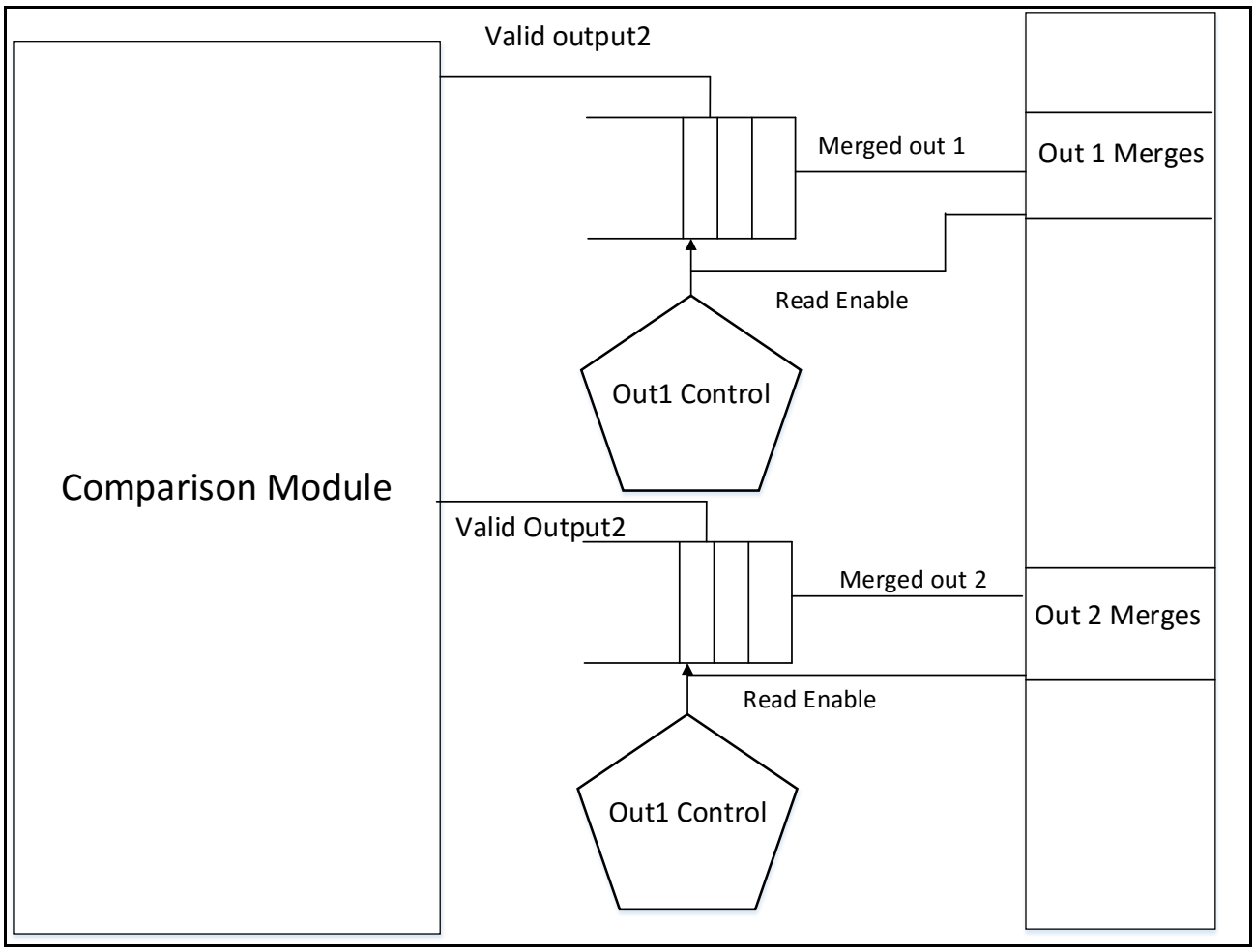


4.3.3 System's output (SGout)

The ScaleJoin module, as described briefly in the previous section, has 2 outputs, one for each stream processed. These outputs are sent to FIFOs pretty much alike the ones described in section 4.3.1. In this section, we are focusing in the way our whole implementation works in order to store the output merged tuples.

To begin with, we are using two FIFOs, one for each output, in order to store the output tuples. Valid output signal enables those FIFOs to write the result. Again, same as the 4.3.1, we use memory controllers, but we use them to extract the output of those FIFOs and store it in Convey's memory. This is exactly the opposite procedure of the one used to input the attributes of each stream. Again, we use a malloc-like Convey's instruction in C, to occupy memory addresses to store the output. Those addresses are inserted into our implementation and used by the memory controllers in order to store each merge found in the occupied addresses of each array. After the whole process is finished, we save each array in a txt file in the project's directory. The output files names are "outX_Y", where X is the number of output of the ScaleJoin algorithm (merged output 1 or merge output 2, figure 4.10) and the Y value shows the processing second in which those merges occurred. Figure 4.11 shows the final implementation of ScaleGate out.

Figure 4.11: ScaleGate out



4.3.4 Scalejoin's Control

To sum up this section, we make a short reference to the whole top level control. Knowing all the pre mentioned modules, it is easy enough to understand the way the control module works. To begin with, all memory controllers mentioned in 4.3.1 send an idle signal to the control module which is high when their respective FIFOs have sent to the ScaleJoin module all attributes needed for the comparisons. Furthermore, the ScaleJoin module also has an idle signal, which is high when no comparison occurs and the result FIFOs are empty. Figure 4.12 shows that also memory controllers, which are controlling the output, have an idle signal which is sent to the control. Below, there is a presentation of each state of this control, which is actually an FSM, and explain on how each state changes and how we manage to preserve the algorithm's challenges.

State 1: Idle State: Waiting for the start signal to procced.

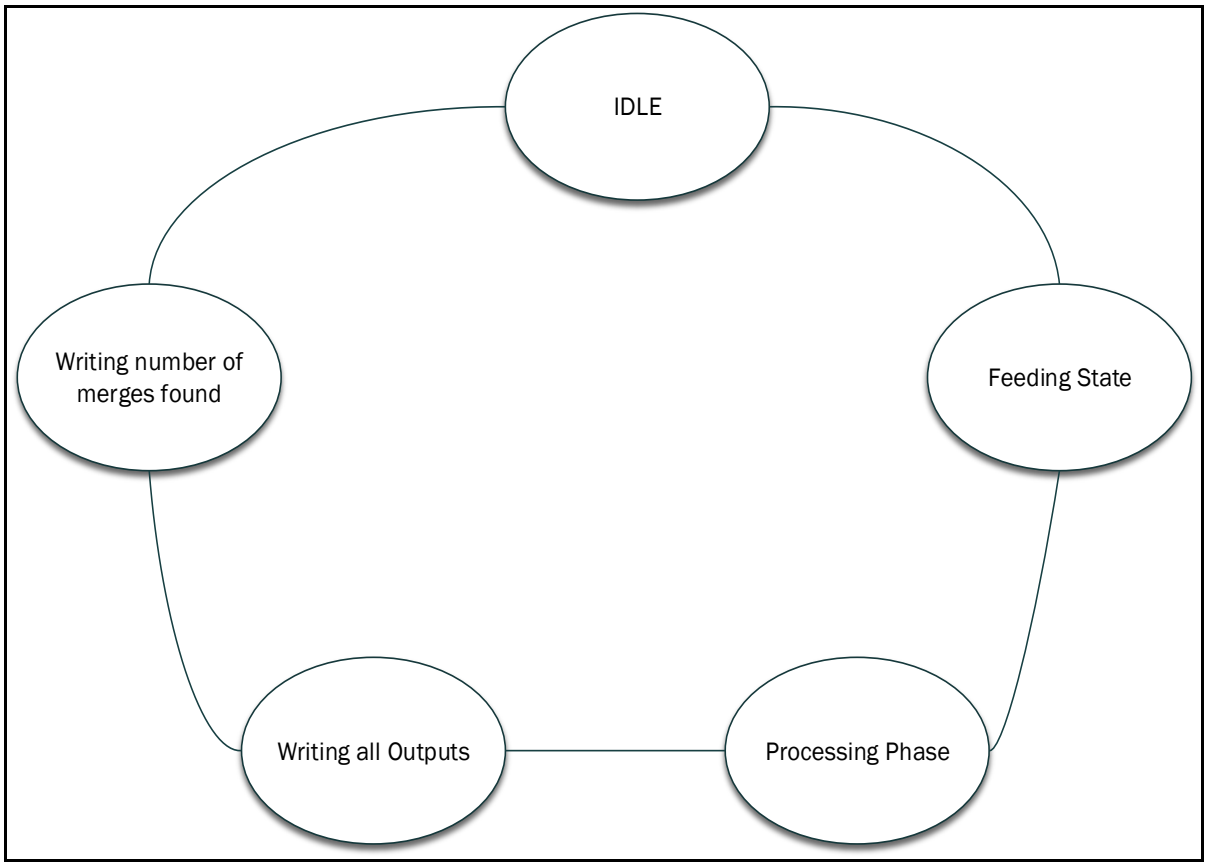
State 2: Feeding state: Feeding ready tuples by checking each streams attributes if they are ready to get processed. Waiting until all tuples are sent to the processing unit.

State 3: Processing Phase: Waiting for the ScaleJoin algorithm to complete all comparisons. Waiting until the Comparison module gets idle.

State 4: Writing Outputs: Waiting until the output FIFOs are empty (Figure 4.11).

State 5: Stores the number of comparisons found and becomes Idle.

Figure 4.12: ScaleJoin Control



4.4 Comparison modules

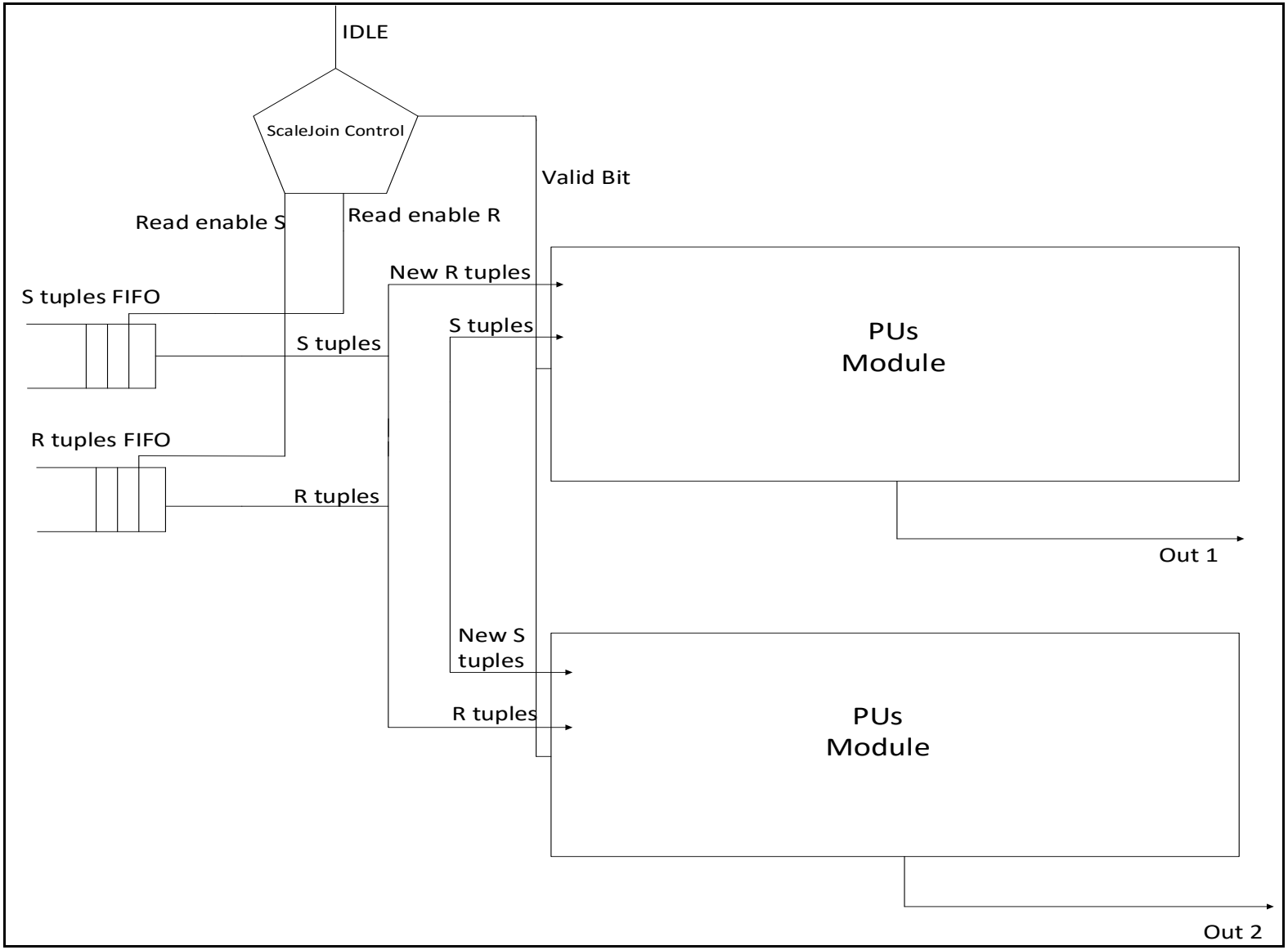
In this section, we make a top-down presentation of each module used in Scalejoin's architecture. Furthermore, we discuss why we choose to each module in our implementation, where parallelization techniques have been applied and what part of the algorithm each module simulates in order to have an accurate implementation with exactly the same outcome, using the same data set in our implementation and in the ScaleJoin algorithm, regardless the data set we are using.

4.4.1 Comparison module architecture

As we move towards our implementations top level, we need to think how all of these modules would work together, in terms of timing all pre mentioned components. We decided to use the architecture, which is shown in the figure 4.13 and we explain how all these modules work with each other and produce the appropriate result. As our implementation starts, we constantly load tuples from each stream, in order to load them to the appropriate set of processing units. To do that, our control has to handle the input FIFOs for each stream and each set in order to load correctly. So, the control module has as input the state bits (empty, full) from each FIFO and keep loading tuples until all tuples in each set are stored. In order to know when all new tuples from both streams are loaded, we use counters. When those counters reach a specific value, which is inputted by the initial C code, a finish signal informs our control that the Load state has finished. Then, FIFOs get the older (in terms of timestamp) tuples each stream has and loads them in the opposite set of processing units (e.g. if, in load state, the new R tuples are stored in the first set, then, in compare state, we feed the first set of processing units with older S tuples) in order to compare all tuples in a stream with all tuples in the opposite stream. Moreover, control module needs to stall all the process if at least one of the processing unit's FIFO is full, that no result merged tuple is overwritten and have all the results outputted. Having this on mind, we have to adjust our processing control and each module separate to block every write or read to each module. Still, as we mentioned in the previous section, each clock cycle, we are searching and reading each processing unit's output round robin like, with a view not to hold our implementation stalled for too long, if a stall take place in one or both sets. In order to define when we finished sending all older tuples to each set, we use to count until it reaches the window size*rate value, which informs us that we have sent all tuples needed for this second comparisons. If one of the sets finishes all the comparisons before the other one, it stalls and waits for the other to finish and then they proceed together in the starting state of the algorithm in order to reload the registers and start the comparison process again.As we mentioned in this thesis, our main goal, regardless of our processing throughput is to make an implementation that is exactly accurate and fulfils Scalejoin's challenges, as presented in its report. So, as we explained in all of the above sections of this chapter, our implementations inputs timestamp sorted tuples with, which in each second we can accept a floating number of incoming tuples per second (e.g. in the n-th second we had as input k tuples, the (n+1)-th second we has as input L tuples e.t.c. with a maximum number of tuples per sec equal to our maximum throughput), makes all the appropriate comparisons that have to be done between the each streams tuples, with caution not to have duplicate compares of the same tuples and while the comparisons take place we output the timestamp sorted merged tuples, exactly as the report describes.

Figure 4.13 below shows the final implementation for the Comparison module, which just inputs tuples to the input FIFOs and subsequently does the pre mentioned processing.

Figure 4.13: Comparison module architecture



4.4.2 Process control module

Moving down on our architecture's hierarchy, we need to refer how our modules are controlled and how they are clocked to work as the ScaleJoin algorithm. In order to do that we have to define, at first, the stages of our algorithm and what the algorithm does in each one of them.

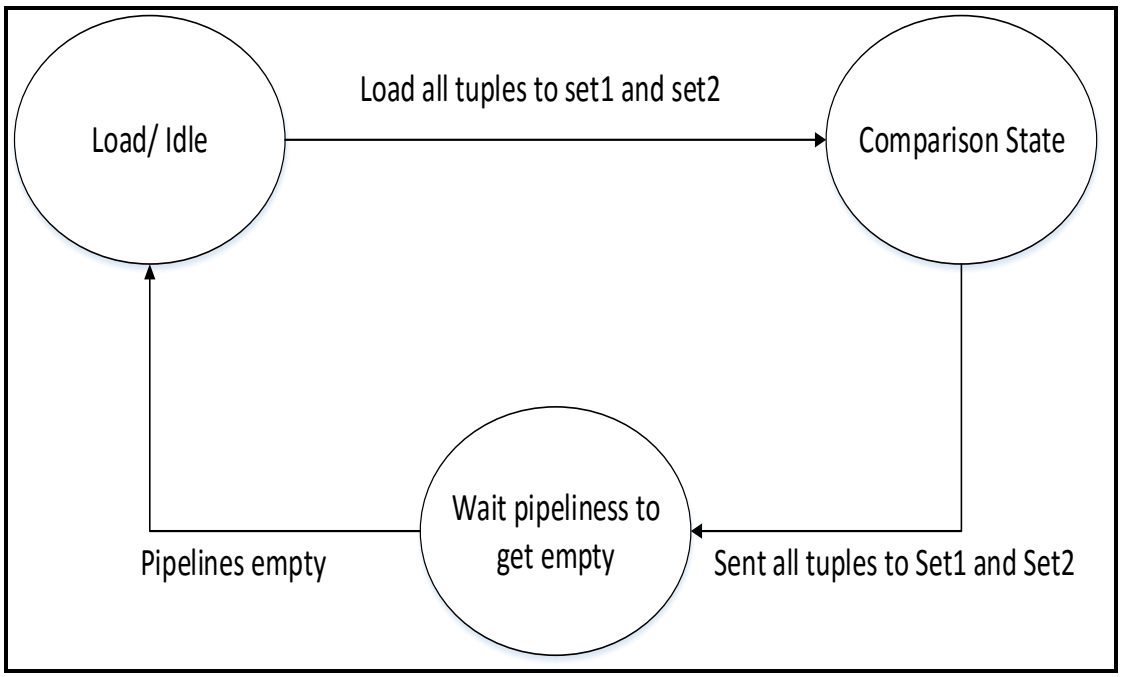
As we mentioned before, our implementation needs to get fed constantly by incoming tuples. Therefore, in our first stage of processing, we must feed the new ready tuples, regardless of the incoming rate, to the processing units. In order to do that, we need to control our input, with a view to insert the amount of tuples that came in the exact second of our processing time, and store them (Tuple S figure 4.16) in the appropriate Processing unit as described in section 4.3.2 and 4.3.1. Also, another thing that we should have in mind is that the modules need to function correctly and have the same outcome as the ScaleJoin algorithm, in terms of its challenges, regardless of the tuples latency (e.g. network latency) and keep our implementation deterministic. In order to do that, we need to stall all of our modules, until all new tuples from both streams are stored in the processing units and all merged tuples written in the previous process must have been outputted from our implementation, in order to proceed to the processing phase of our algorithm.

In the next stage, we have to process and compare the new tuples stored in the first phase with the tuples came a window size before. As we mentioned in the beginning, a window size is a time based (seconds) variable which defines how much older, a valid comparison, can be. In other words, in order to define if a comparison is being accepted, in terms of window size, or not, we need to subtract the timestamp values of the compared tuples and the outcome should be less or equal than the window size variable. When the second phase starts, we feed the Set of N processing units with older tuples that have to be compared with the new ones. To do that, in each cycle, we draw a tuples from the other Stream and store it in the first processing unit (Tuple R Figure 4.16). In the same time, we need to "inform" the first processing unit that this comparison is valid, in terms of window size, and store the value "1" in the valid register (Figure 4.16). In the next cycle, the value sent before, is stored in the next processing unit and the next one is drawn from the stream. Therefore, as we explained in section 4.3.2, the older tuples that are fed in this phase, are moving in a pipeline fashion and so we manage to compare all tuples from the one stream with all the new ones in the other stream. Each successful merge is stored to the FIFO of each Processing unit until a permission is given to the processing unit to output its merged tuple, as we explained in section 4.3.1. Something that is worth mentioning is that in each phase of the process, the output control module (Figure 4.15) is searching each processing unit for new merged tuples. This processing phase continues until all of the older tuples are sent to the appropriate set of processing units and then we move to the next stage of processing.

As we move to the third and final stage of our implementation's processing, we need to wait for our pipeline to empty. The reason we used this stage is that when all older tuples are fed to a set of processing units, does not mean that all the comparisons have been made (!). So, in order to have all the comparisons done, we have to wait until the last tuple fed in stage 2 reaches the final pipelined processing unit and then call back the first state of our control and proceed with the next load in each Processing unit.

As someone can easily understand, this module is an FSM, which has as inputs the output signals of its controlling components. The output signal of each component define in which state each component is, and decide, given the combination of all of those signals, what each module must do in order to keep our whole implementation up to the challenges of the ScaleJoin algorithm, which is our main goal. The figure 4.14 below shows a brief diagram of our FSM and its functionality.

Figure 4.14: Comparison Control

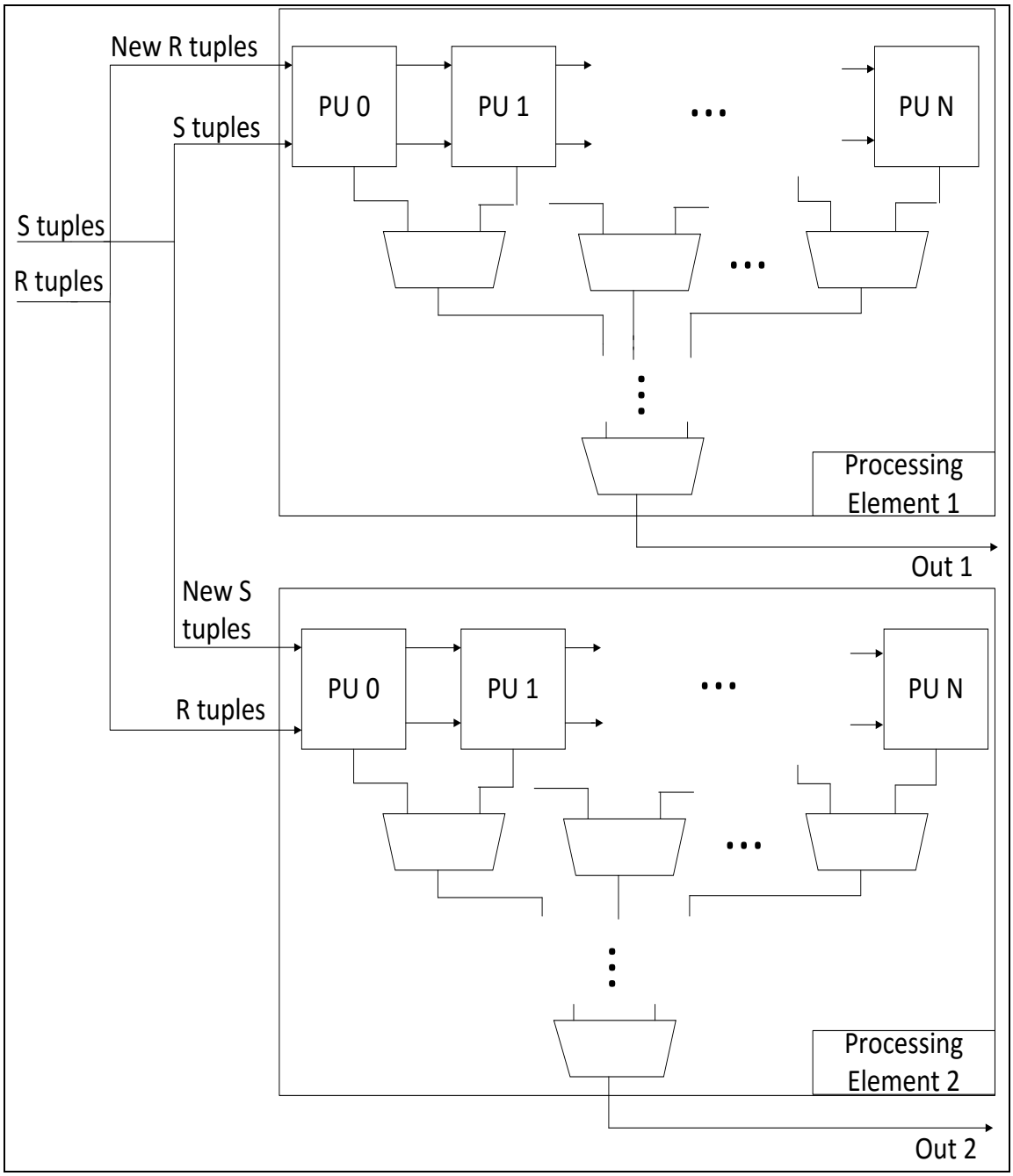


4.4.3 Parallelization of processing units

Join processing algorithms have a need in processing throughput, which is constantly increasing. In order to achieve higher throughput, ScaleJoin algorithm is using many processing units to increase its throughput, via the parallelization of the whole processing on tuples given and therefore parallelization upon the overall comparisons. So, in order to decide which parallelization technic we are going to use to our processing units, we need to think about satisfying the parallelization challenges of the ScaleJoin algorithm and furthermore, as it is natural, have exactly the same outcome as the algorithm itself would have if the same data sets were given as an input in the algorithm.

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 4.15 shows a set of processing units. A set of processing units is a number of processing units which are connected to each other in a pipeline fashion in order to achieve the sliding window having a sliding window, as mentioned in the report [7].

Figure 4.15: Reconfigurable StreamJoin Architecture



As it comes to the functionality of each set, each cycle, we load each processing unit with 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 streams which have a maximum timestamp difference of a Window size, in order to make all the appropriate comparisons that need to be done in a processing second. The opposite tuples are stored in one processing unit, they are processed and then they move to the next processing unit in a pipeline fashion, with a view to comparing all old tuples with all new ones (sliding window). So, in the end of each cycle, each comparison has been made and the tuples that need to be merged are stored in the respective processing unit's FIFO, as it will be described in section 4.3.4, which focuses on the architecture of a processing unit. Thus, all the possible correlations among the newly arrived tuples and the previously arrived tuples are computed. Also,

each result passes through a network of MUXes as an output. When all tuples from the processing window are streamed and no other results have to be sent out, then the processing 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 PEs, as we will show below.

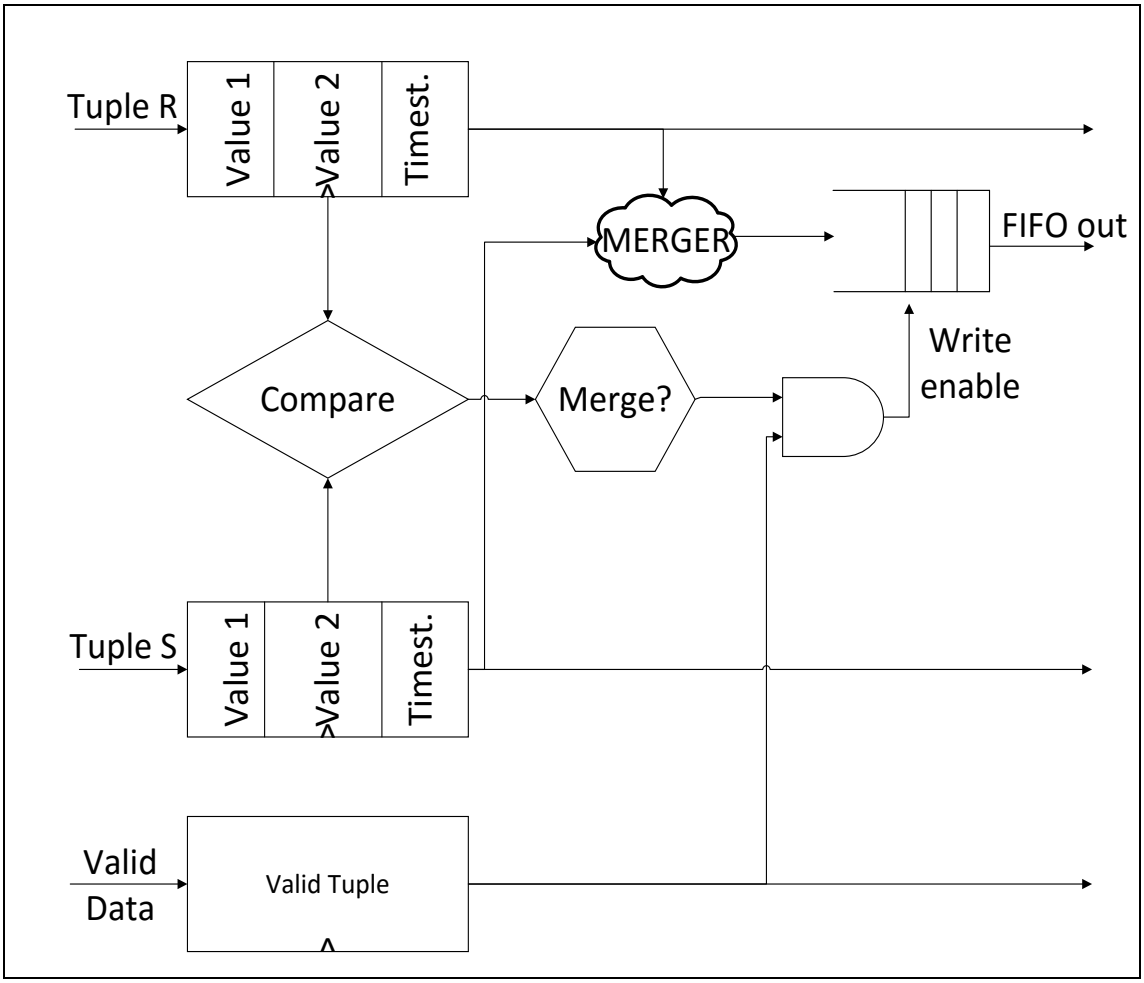
4.5 Processing Unit

As we referred in previous chapters, processing units are the fundamental part of the ScaleJoin algorithm. The processing units need to compare one tuple of each stream and create a new tuple, if we have a successful compare, by merging the tuples into one that has all the attributes of both tuples. This module is being fed tuples from stream R and stream S, compares their attributes and defines if a compare is successful and then proceeds to the merging phase of the tuples.

In order to create a modules 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 is presented in the next section (section 4.3.2).

Our final implementation for this module is shown in figure 4.16. This module has two 96-bits registers, each one for each merging stream's tuple, and each of those tuples have two attributes 32-bits each and an ID/timestamp to define each comparison between those tuples. The value of those registers is being driven into a compare module, which is responsible for the merge-or-not decision. The merge-or-not is set in the top level of our architecture. In our implementation, we used a bound given in the ScaleJoin report [7] in which we decide that we have a successful compare when the absolute difference of the first attribute of each tuple and the absolute difference of the second attribute of each tuple is less than offset number 10. The outcome of the compare module is a 64-bit signal, which is a concatenation of the ID/timestamp of the tuples merged. In addition, we have another 1-bit register, which works as a validation bit, and its purpose of use is to define which comparison is valid. When we have a successful compare and we have a valid comparison then the FIFO is being enabled in order to write the outcome of the comparison. Each processing unit has its own FIFO, which has depth equal to 4, and each cycle we know in which state the FIFO is. FIFO is enabled by other modules in order to read a value and for each value which has as outcome, it returns a valid bit, in order to decide which value is a valid compare or no

Figure 4.16: PU architecture



4.6 Conclusion

All in all, the above sections describe the whole implementation of the ScaleJoin in each FPGA. As we mentioned before, every copy in each FPGA runs in parallel with the others and the process in each one is completely irrelevant with the others. This is made to make the most of the hardware given by Convey HC-2 and, at the same time, in each FPGA we used this architecture in order to process each stream as efficiently as we could.

Chapter 5: Evaluation

5.1 Introduction

This chapter presents the performance results of the proposed system. Firstly, we present how we made sure that our implementation is totally correct and has the same output, given the same input, as the initial ScaleJoin algorithm. Then, we present the performance bounds for a system that implements the stream join processing, as presented in [6]. Next, we show the performance upper limits, in terms of comparisons/sec and tuples/sec, that the proposed hardware solution can offer. Last, an evaluation benchmark among other related works that are presented in bibliography and accelerate the stream join processing with our proposed system is presented.

5.2 Cross-check evaluation

Our first step during the evaluation phase is to make sure that we are made a completely accurate implementation, which results in the same outcome as the initial algorithm, given the same input. At this point, with much help from Chalmers University of Technology and more specifically from Gulisano, Vincenzo, Nikolakopoulos, Y., Papatriantafilou M. and Tsigas F., a data set has been sent to us, which represent the input streams and their respective merge results. Then, we decided to create a non-parallel C code for ScaleJoin, taking into account the things said in [7]. Our next step is to import each tuple of each stream in this code and make sure our results are the same as the results, which are sent from Chalmers University. Finally, after making sure that the C code is an accurate copy of the initial ScaleJoin algorithm, we had to make sure that our FPGA architecture has the same output as the pre mentioned C code and also with the initial algorithm. As a result, each random tuple created for each stream created by the C code, which controls Convey HC-2, was stored in 2 txt files. Those txt files were then loaded to the copy of the ScaleJoin algorithm and made sure that each random run, returned the same results.

5.3 Theoretical Performance bounds

The stream join processing includes the comparison of the newly arrived tuples from each one of two streams with all the in time-window tuples of the other stream. Considering that the tuples from both streams arrive with a rate T tuples/sec and the time processing window has size W, then the total number of comparisons that need to take place at each second is about $2 \times W \times T^2$ [6]. Thus, according to the above formula we can calculate the maximum number of the comparisons that can be implemented by a system per second, given the window size and the maximum throughput rate in tuples per second that the system can process.

5.4 Experimental setup

We use the same benchmark that was used for evaluating ScaleJoin [6], CellJoin [10] and Handshake joins [7, 8]. The R tuples have 4 attributes (t_s,x,y,z), where x,y,z are of types int, float and char[20], respectively. The S tuples also have 4 attributes (t_s, a, b, c, d), where a, b, c, d are of types int, float, double and bool, respectively. The values for attributes x, y, a and b are random using a uniform distribution in the interval [1−10,000]. An output tuple (t_s, x, y, z, a, b, c, d) is created for each pair of tuples t_R, t_S only if the below comparisons are true:

$|t_S.a - t_R.x| <= 10$ and $|t_S.b - t_R.y| <= 10$.

We use a C code, which generates random tuples, according to the above interval, for both streams and stores them into Convey's RAM at each second. Next, the new tuples of R and S streams are loaded to the reconfigurable part of the ScaleJoin module, while the older tuples are streamed for processing. Last, the processor reads and presents the stream join results at each 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.

5.5 Performance Evaluation

This section presents the performance achieved by the proposed reconfigurable system. Also, we compare the performance achieved by our solution vs. the performance achieved by the most performance efficient multicore solution, as presented in [6]. Last, we show the scalability of our hardware-based solution when a single and four parallel FPGA devices of a Convey HC-2ex server are used.

As referred above, the reconfigurable architecture was mapped on a Convey HC-2ex server. The HC-2ex server is equipped with four Virtex 6 LX760 FPGA devices and a 4-core Intel Xeon CPU at 2.13 GHz with 24GB RAM. The resource utilization for the proposed architecture is presented in Table 1; the processing system is clocked at 150 MHz. As Table II shows, the proposed architecture uses less than 30% of the available resources. In our future plans, we aim at proposing a novel architecture taking advantage of even greater parallelism, thus, getting better performance results.

Table 5.1: Resource utilization for hardware-based stream join architecture

FPGA Resources	**System's FPGA Utilization**
DSPs	0/864 (0%)
Slice Registers	158777/948480 (16%)
Slice LUTs	151688/474240 (31%)
Registers used as Memory	22.904/132480 (17%)
Block RAMs	114/720 (15%)
Number of IOBs	909/1200 (75%)

On the other hand, the software-based reference system, as presented in [6], is equipped with a 2.6 GHz AMD Opteron 6230, 48 cores over 4 sockets and 64 GB RAM. Both systems' performance was measured evaluating the maximum number of comparisons that can be executed per second and the maximum throughput achieved, i.e. number of tuples per sec., and the measurements are actual, experimental results from runs on the respective platforms.

Figure 1 shows the processing capabilities of the proposed system. According to the performance results presented in [6], software-based reference system with 48 cores can achieve approximately 4 billion comparisons/sec for various window sizes. On the other hand, the proposed FPGA-based system with 1 FPGA device can offer up to 18.5 billion comparisons/sec, while when all the four available FPGA devices are used can offer up to 74 billion comparison/sec. Thus, our proposed solution outperforms in terms of processing the best multi-core solution by a factor of about 4x when a single FPGA is used or by a factor of 19x when four parallel FPGA devices are mapped. We note here that there is no performance lost when the application runs on all four FPGAs due to the intrinsic parallelism.

Figure 2 shows the throughput achieved in tuples per second for both systems. As the results indicate, the FPGA-based solution can process with a single FPGA device 2 times higher throughput streams vs. a multicore solution, as the full reconfigurable system outperforms the fastest software-based multicore solution by at least a factor of 4x.

Figure 5.1: Processing rate (comparisons/sec) for SW-based multicore ScaleJoin and FPGA-based solution ScaleJoin

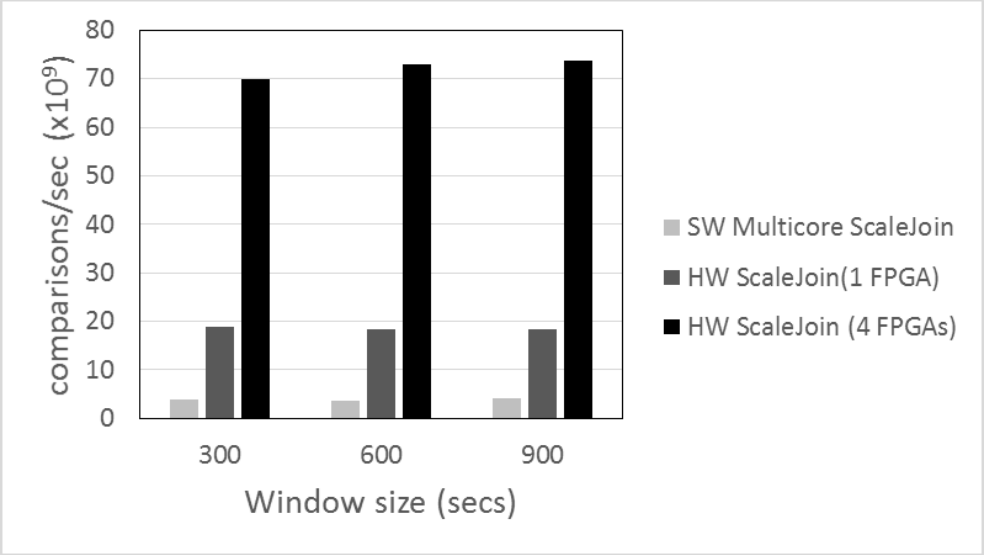
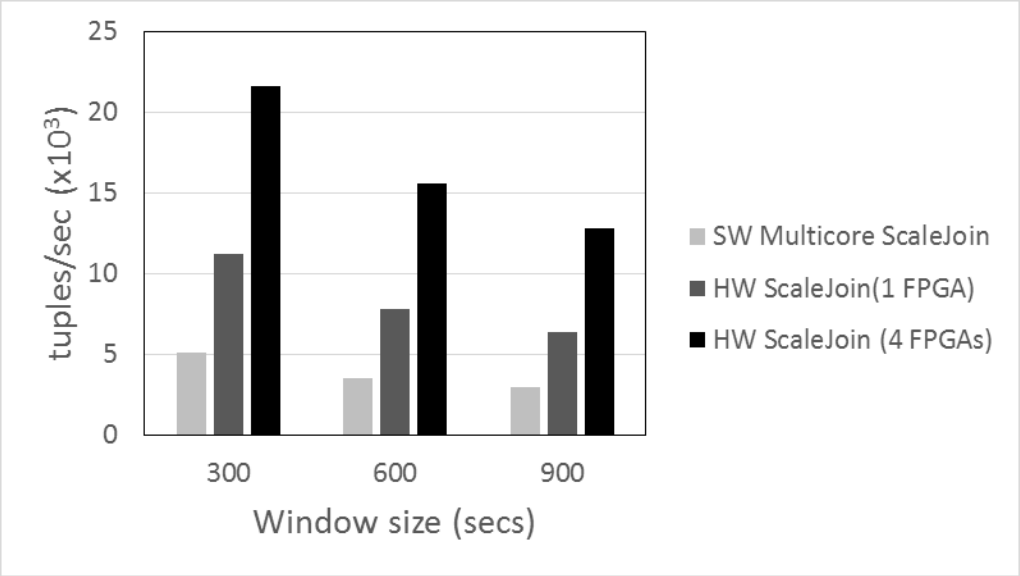


Figure 5.2: Throughput (tuples/sec) for SW-based multicore ScaleJoin and FPGA-based solution ScaleJoin



In addition, it is important to mention the scalability of the hardware proposed solution. According to Figure 2, the performance, when 4 FPGA devices are used, scales proportionally to the square root of the number of parallel processing units, which can be shown from the formula described above, too. This is the theoretical upper bound, and it is realized in practice too.

5.6 Benchmark Performance Evaluation

This section compares the performance of the proposed stream join implementation vs. other state-of-the-art multicore solutions under the same benchmark, which was described above. The comparison takes place in terms of maximum throughput rate that each system can achieve and the maximum number of comparisons that each system can execute. Last, we compare the proposed system vs. previous hardware implementations, despite the fact that there is not a single clear, common benchmark for this case.

As referred above, there are three works on stream join problem that followed the same benchmark. The first work is the CellJoin system [10], where a Cell processor was used for implementing window-based stream joins. The Celljoin system used 9 processors, i.e. one Power PC processing element and 8 Synergistic Processing Elements, for the parallel stream join processing. The second work is the software-based Handshake algorithm [7, 8]. This method was evaluated using an AMD Opteron 6174 "Magny Cours" machine with 48 parallel cores. The last work is about the implementation of the most performance efficient stream join algorithm, ScaleJoin [6]. The performance evaluation of the ScaleJoin algorithm took place on a 2.6 GHz AMD Opteron 6230 with 48 cores.

Table 2 shows the performance achieved by the three above works and the proposed reconfigurable implementation. As the results show, the Handshake system offers the best throughput rate which can reach in a software-based system with about 5125 tuples per sec, among the previously presented systems. On the other hand, the proposed reconfigurable system with a single FPGA device offers a throughput rate up to 6400 tuples per sec, whereas when the full system is used the rate reaches up to 12800 tuples per sec. Thus, our system seems to outperform any other proposed software-based solution by at least a factor of 2x.

The other performance metric, which is widely used, is the number of comparisons per second without taking into consideration the I/O issues. The most performance efficient solution, as presented in Table 2, is the ScaleJoin system. It can offer up to 4 billion comparisons pes sec. On the contrary, the reconfigurable system can offer up to 74 billion comparisons per sec including I/O time. Thus, according to Table 2, the reconfigurable solutions seems to outperform any other state-of-the-art multicore solution by at least one order of magnitude as far as the number of executed comparisons on streaming data.

Lastly, there are some previous works [12, 13 and 14] which map the stream join problem but they do not follow an open source benchmark to compare with. The work in [6] offers really good performance but only for small processing windows, which can reach only up to 1024 tuples. On the contrary, our proposed work offers real-time stream processing without taking into account the size of the time processing window. Next, the work in [13] offers really good performance results but the results are not from real hardware runs but from simulation. On the other hand, the performance results of our proposed solution come from real life experiments under specific real time restrictions. The work in [14] offer performance results which can reach up to 200 million tuples/sec but without giving any more details about the benchmark that the authors used.

Table 5.2. SW multicore stream join vs. FPGA based stream join on Benchmark Evaluation

Systems	Handshake system [7, 8]	ScaleJoin system [6]	CellJoin system [10]	FPGA-based ScaleJoin system
CPU Cores	40	48	9	1 CPU + 4 FPGAs
CPU type	2.2 GHz AMD Opteron	2.6 GHz AMD Opteron	1 PPE and 8 SPEs	2.13 GHz Intel Xeon
Processing Time Window (mins)	15	15	15	15
Max Throughput Rate (tuples/sec)	5125	3000	2000	12800
Max Processing Rate (Comps/sec)	1.5×10^9	4×10^9	-	74×10^9

5.7 Conclusion

As the results show, we outperform, in most occasions by far, the related works that have been done in the stream join branch both in software and hardware. Our implementation can achieve tens of billions of comparisons in second, achieving, at the same time, a great number of maximum new incoming tuples per second. Moreover, regarding the software-based implementation of ScaleJoin, we achieve a high enough 19x of its maximum throughput, while we also achieve 4.5 times greater tuple rate.

Chapter 6: Conclusion and Future work

6.1 Conclusion

This works presented an FPGA-based system that implements a widely used steam data mining operator, i.e. stream join. To the best of our knowledge, this is the first work that maps a stream join operator on a high-end multi-FPGA system, such as the Convey HC-2ex server. Next, we presented and analyzed an efficient, extensible, scalable and generic reconfigurable architecture for the stream join workload. The main characteristics of the proposed architecture are analyzed below:

- Efficient. According to the performance evaluation presented in this work, the proposed architecture seems to outperform any other state-of-the-art published work. The proposed architecture, when mapped on a high-end multi-FPGA system with fast I/O links, can offer about one order of magnitude higher processing rates than any other software or hardware-based state-of-the-art system.

- Extensible. The proposed architecture can be easily mapped in parallel to more than a single FPGA device, as shown in experimental evaluation. We presented the results for implementations when one and four FPGA devices, without loss of genericity, achieving the theoretical expected increase in performance.

- Scalable. The proposed architecture offers high fine grained scalability. As shown, increasing the number of available PUs in each FPGA device, it can offer even better performance results.

- Generic architecture. The proposed architecture is generic and can be used to tackle the stream-based workloads on reconfigurable hardware. As shown, the processing that takes place into the main processing unit of the system can change according to the mapped application. Thus, the proposed architecture can easily be extended or used to any type of streaming processing that combines data from different streams.

To conclude, FPGA's are particularly well suited for this form of computation vs. software-based multicore solutions, but fast I/O and the proper memory organization is necessary in order to fully realize these advantages.

6.2 Future work

Regarding future work, we are aware of certain adjustments that can significally increase our processing throughput, while we keep our implementation accurate. As shown in Table 5.1, we occupy less than a third of the total LUTs of Virtex 6 LX 760. This is because we faced certain problem, while we were mapping a larger implementation into the bitfile. Should this problem is solved, we would be capable of mapping more PUs in each set of processing units and so we would load a higher number of new tuples in each run. Then, we could also organize our stalls in each set

of processing units. We could create smaller groups and stall those smaller groups dynamically and not the whole set. Lastly we can achieve a greater clock period (FPGAs maximum is 200 MHz) and improve our results.

References

[1] Fayyad, U. M., Piatetsky-Shapiro, G., & Smyth, P. (1996, August). Knowledge Discovery and Data Mining: Towards a Unifying Framework. In KDD (Vol. 96, pp. 82-88).

[2] Gibbons, P. B. (2015, May). Big data: Scale down, scale up, scale out. In Parallel and Distributed Processing Symposium, 2015. IPDPS 2015. IEEE International (p. 3).

[3] Xie, J., & Yang, J. (2007). A survey of join processing in data streams. In Data Streams (pp. 209-236). Springer US.

[4] Kang, J., Naughton, J. F., & Viglas, S. D. (2003, March). Evaluating window joins over unbounded streams. In Data Engineering, 2003. Proceedings. 19th International Conference on (pp. 341-352). IEEE.

[5] Halstead, R. J., Sukhwani, B., Min, H., Thoennes, M., Dube, P., Asaad, S., & Iyer, B. (2013, April). Accelerating join operation for relational databases with FPGAs. In Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual International Symposium on (pp. 17-20). IEEE.

[6] Gulisano, V., Nikolakopoulos, Y., Papatriantafilou, M., & Tsigas, P. ScaleJoin: a Deterministic, Disjoint--Parallel and Skew--Resilient Stream Join. In Big Data (Big Data), 2015 IEEE International Conference. IEEE.

[7] Teubner, J., & Mueller, R. (2011, June). How soccer players would do stream joins. In Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (pp. 625-636). ACM.

[8] Roy, P., Teubner, J., & Gemulla, R. (2014). Low-latency handshake join.Proceedings of the VLDB Endowment, 7(9), 709-720.

[9] Sadoghi, M., Javed, R., Tarafdar, N., Singh, H., Palaniappan, R., & Jacobsen, H. A. (2012, April). Multi-query stream processing on fpgas. In Data Engineering (ICDE), 2012 IEEE 28th International Conference on (pp. 1229-1232). IEEE.

[10] Gedik, B., Bordawekar, R. R., & Philip, S. Y. (2009). CellJoin: a parallel stream join operator for the cell processor. The VLDB journal, 18(2), 501-519.

[11] Oge, Y., Miyoshi, T., Kawashima, H., & Yoshinaga, T. (2012, September). Design and implementation of a merging network architecture for handshake join operator on FPGA. In Embedded Multicore Socs (MCSoC), 2012 IEEE 6th International Symposium on (pp. 84-91). IEEE.

[12] Oge, Y., Miyoshi, T., Kawashima, H., & Yoshinaga, T. (2011, November). An implementation of

handshake join on FPGA. In Networking and Computing (ICNC), 2011 Second International Conference on (pp. 95-104). IEEE.

[13] Qian, J. B., Xu, H. B., DONG, Y. S., Liu, X. J., & Wang, Y. L. (2005). FPGA acceleration window joins over multiple data streams. Journal of Circuits, Systems, and Computers, 14(04), 813-830.

[14] Oge, Y., Yoshimi, M., Miyoshi, T., Kawashima, H., Irie, H., & Yoshinaga, T. (2013, December). An Efficient and Scalable Implementation of Sliding-Window Aggregate Operator on FPGA. In Computing and Networking (CANDAR), 2013 First International Symposium on (pp. 112-121). IEEE.

[15] Mueller, R., Teubner, J., & Alonso, G. (2009). Streams on wires: a query compiler for FPGAs. Proceedings of the VLDB Endowment, 2(1), 229-240.

[16] Deligiannakis A., Kotidis Y. and Roussopoulos N. (2004) Hierarchical in-Network Data Aggregation with Quality Guarantees. Proceedings of the 9th International Conference on Extending DataBase Technology.

[17] Babcock B., and Olston C. (2003) Distributed top-k monitoring. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

[18] Domingos P. and Hulten G. (2000) Mining High-Speed Data Streams. In Proceedings of the ACM KDD Conference.

[19] Hulten G., Spencer L., Domingos P. (2001).Mining Time Changing Data Streams. ACM KDD Conference.

[20] AslamJ., Butler Z., Constantin F., CrespiV., CybenkoG. and RusD. (2003) Tracking a Moving Object with a Binary Sensor Network. In Proceedings of ACM SenSys.

[21] Bonnet P., Gehrke J., and Seshadri P. (2001) Towards Sensor Database Systems. In Proceedings of the 2nd International Conference on Mobile Data Management, London, UK.

[22] Deligiannakis A., Kotidis Y. and Roussopoulos N. (2004) Compressing Historical Information in Sensor Networks. In Proceedings of the ACM SIGMOD International Conference on Management of Data.

[23] Deshpande A., Guestrin C., Hong W. and Madden S. (2005) Exploiting Correlated Attributes in Acquisitional Query Processing. In Proceedings of the 21st International Conference on Data Engineering.

[24] Ganesan D., Greenstein B., Perelyubskiy D., Estrin D. and Heidemann J. (2003) An Evaluation of Multi-Resolution Storage for Sensor Networks. In Proceedings of ACM SenSys.
[25] Hellerstein J.M., HongW.,Madden S. and Stanek K. (2003) Beyond Average: Toward Sophisticated SensingwithQueries. InternationalWorkshop on Information Processing in Sensor

Networks.

[26] Ali, M. H., Mokbel M. F., Aref W. G. and Kamel I. (2005) Detectioin and Tracking of Discrete Phenomena in Sensor-Network Databases. In Proceedings of the 17th International Conference on Scientific and Statistical Database Management.

[27] Chu M., Haussecker H. and Zhao F. (2002) Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. International Journal of High Performance Computing Applications.

[28] Silverman B.W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman and Hall.

[29] Cerpa A., Elson J., Estrin D., Girod L., Hamilton M. and Zhao J. (2001) Habitat monitoring: Application driver for wireless communications technology. In Proceedings of ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean.

[30] Chen W., Hou J. C. and Sha L. (2003) Dynamic Clustering for Acoustic Target Tracking in Wireless Sensor Networks. In Proceedings of the 11th IEEE International Conference on Network Protocols.

[31] Guestrin C., Bodik P., Thibaux R., Paskin M. and Madden S. (2004) Distributed Regression: an Efficient Framework forModeling Sensor Network Data. In Proceedings of the 3rd International Conference on InformationProcessing in Sensor Networks.

[32] Halkidi M., Papadopoulos D., Kalogeraki V. and Gunopulos D., (2005) Resilient and Energy Efficient Tracking in Sensor Networks. International Journal of Wireless and Mobile Computing.

[33] SAMOA Developer's Guide (Scalable Advanced Massive Online Analysis) ARINTO MURDOPO, ANTONIO SEVERIEN, GIANMARCODEFRANCISCI MORALES, AND ALBERT BIFET Yahoo! Labs Barcelona.

[34] Yasin Oge, Takefumi Miyoshi, Hideyuki Kawashima, Tsutomu Yoshinaga: A fast handshake join implementation on FPGA with adaptive merging network. SSDBM 2013: 44

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