平成２４年度修士論文

# Design and Implementation of a Hardware Accelerator for Handshake Join on FPGA


大学院情報システム学研究科

情報ネットワークシステム学専攻

学 籍 番 号　：　１１５２００７

氏　　　名　：　オゲ　ヤースィン

主任指導教員　：　吉永　努 教授

指 導 教 員　：　入江　英嗣 准教授

指 導 教 員　：　森田　啓義 教授



提出年月日　：　平成２４年７月２０日

(表紙裏)

# Contents


**1 Introduction** **1**
1.1 Background . . . 1
1.2 Motivation . . . 1
1.3 Objectives . . . 2

**2 Related Work** **3**
2.1 Data Processing on FPGA . . . 3
2.2 Window Join Implementation on FPGA . . . 3
2.2.1 Window Join . . . 3
2.2.2 Implementation of Window Join . . . 6
2.3 Other works . . . 8

**3 Design of Handshake Join** **9**
3.1 Handshake Join . . . 9
3.2 Design Issues of Handshake Join Hardware . . . 11
3.3 Design Strategy of Handshake Join Hardware . . . 13

**4 Implementation of Handshake Join Architecture** **14**
4.1 Join Core . . . 14
4.2 Merger . . . 18
4.3 Merging Network . . . 19
4.4 Admission Control . . . 20
4.5 Evaluation of the Handshake Join Hardware . . . 21
4.5.1 Resource Usage and Signal Delay . . . 21
4.5.2 Performance Evaluation . . . 22

**5 Discussion on Buffer Size Tuning** **25**
5.1 Static Tuning . . . 25
5.2 Adaptive Tuning . . . 27

**6 Design and Implementation of Adaptive Merging Network** **30**
6.1 Design Overview . . . 30
6.2 Implementation of the Proposed Architecture . . . 32
6.2.1 Join Core . . . 32
6.2.2 Adaptive Merging Network . . . 32
6.2.3 Admission Control . . . 34
6.3 Evaluation of the Proposed Implementation . . . 34
6.3.1 Resource Usage and Signal Delay . . . 35

6.3.2 Resource Usage Comparison . . . . . 36
6.3.3 Performance Evaluation . . . . . 39

**7 Conclusion** **42**
7.1 Summary . . . . . 42
7.2 Future Work . . . . . 43

**Acknowledgement** **44**

**References** **46**

**List of Publications Related to the Thesis** **47**

# List of Figures


2.2.1 Window join (figure adopted from [1]). . . . . . . . . . . . . . . . . . . . . . . . . . . 5

3.1.1 The basic idea of handshake join (adopted from [1]). . . . . . . . . . . . . . . . . . . . 9
3.1.2 Handshake join with tuple-based windows. Tuples from the two input streams are flowed in opposite direction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
3.1.3 The semantics of handshake join (adopted from [1]). . . . . . . . . . . . . . . . . . . . 10
3.1.4 Parallelization of handshake join operation. Each of the cores evaluates its own segment of the both windows. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.2.1 Overview of the handshake join architecture with tuple-based windows for FPGA implementation (adopted from [1]). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

4.1.1 State transition diagram of the join core circuits. . . . . . . . . . . . . . . . . . . . . . 16
4.1.2 Immediate scan strategy. When a tuple from input stream $R$ (a) or $S$ (b) enters to the join core $k$, the immediate comparison will be triggered respectively in the same join core (figure adopted from [1]). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
4.2.1 State transition diagram of the merger circuits. . . . . . . . . . . . . . . . . . . . . . . 18
4.3.1 Binary tree-like connection. An example of a result-merging network for 16 join cores. . 19
4.3.2 Connection between join cores and merging network. . . . . . . . . . . . . . . . . . . 20
4.5.1 Maximum clock frequency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.5.2 Overall resource consumption. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
4.5.3 Evaluation of the handshake join architecture. . . . . . . . . . . . . . . . . . . . . . . 23
4.5.4 Maximum input throughput. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

5.1.1 Results of the simulation for different buffer sizes ($2^i$ *where* $i = 2, \ldots, 11$) at 100% match rate. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26
5.1.2 Results of the simulation for four different configurations at 20%, 40%, 60%, 80% and 100% match rates. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27
5.2.1 Results of the simulation for input streams of three different characteristics at 20% and 40% match rates. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29

6.1.1 Adaptive merging network with four bufferless join cores. . . . . . . . . . . . . . . . . 31
6.2.1 The connections of the ring structure in the proposed merging network. . . . . . . . . . 33
6.3.1 Maximum clock frequency. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.3.2 Overall resource consumption. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
6.3.3 Comparison of slice registers and slice LUTs utilization between the baseline implementation (presented in Chapter 4) and the proposed implementation (presented in Chapter 6). 36
(a) Slice registers. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
(b) Slice LUTs. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

6.3.4 Comparison of occupied slices and BRAM utilization between the baseline implementation (presented in Chapter 4) and the proposed implementation (presented in Chapter 6). 38
(a) Occupied slices. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
(b) Block RAM. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
6.3.5 The number of result tuples, and the total number of cycles required to complete the operation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
6.3.6 Input throughput with 64 join cores. . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

# List of Tables

2.1 Definition of terms used in cost model . . . . . . . . . . 6

4.1 Specifications of XC6VLX240T-1 . . . . . . . . . . 21
4.2 Hardware resource usage . . . . . . . . . . 22

5.1 Buffer size configurations . . . . . . . . . . 26

6.1 Hardware resource usage . . . . . . . . . . 35

# Chapter 1

# Introduction

## 1.1 Background

Nowadays, stream data processing systems demand much more functionality than what was available in the past. Many data processing tasks, such as financial analysis, traffic monitoring and data processing in sensor networks, are required to handle a huge amount of data with certain time restrictions. Low-latency and high-throughput processing are key requirements of systems that process unbounded and continuous input streams rather than fixed-size stored data sets.

Most of the modern relational databases (*Database Management Systems, DBMSs*) have been added superfluous features. All of them should provide basic set operations including union, intersection, difference and Cartesian product. Moreover, they support other operations such as join, selection, projection and division. Likewise, stream databases (*Data Stream Management Systems, DSMSs*) also support similar operations. One of these fundamental operations is called stream join or window join [2] that introduces window semantics besides value-based join predicates.

Stream databases deal with infinite streams of data that have to be processed immediately for real-time applications. It is stated in [2] that, theoretically, processing a join over unbounded input streams requires unbounded memory since every tuple in one infinite stream must be compared with every tuple in the other. It is clear that this causes practical problems. To solve the problem, the window semantic is introduced for practical applications. That is to say, a finite subset of the unbounded input data is defined as a *window* for each input stream, and a join predicate is evaluated over the windows.

## 1.2 Motivation

Teubner and Mueller have provided new insight into stream join algorithm, and proposed a novel approach, namely *handshake join*. It is a stream join algorithm that can support very high degrees of parallelism and attain unprecedented success in throughput speed [1]. They demonstrate a software implementation using a modern multi-core CPU. It considerably outperforms CellJoin [3], which is another well-known implementation of window-based join for the Cell processor. They also mention that handshake join can naturally leverage available hardware parallelism even though a complete hardware design of handshake join is not provided in [1].

Handshake join enables us to parallelize the matching process in a very elegant way; however, there is a practical problem of the approach: results of parallel processes should be collected and merged into a single output stream. In addition, the parallel execution of joins can result in a higher output rate than a sequential execution because the same number of results are produced in a shorter time. In other words, a larger number of results can be produced per unit time, and the merging process would be quickly

overloaded. This is the case, for example, with such applications as *TCP SYN Flood* detection [4] where a volume of output may be instantaneously generated, depending on the dynamic characteristics of input streams. Following design issues should be taken into account when it comes to implementing handshake join hardware:

1. a scalable mechanism (in terms of the resource usage and the signal delay) that merges results into a single output stream,
2. a flow control mechanism (between all join cores and the output port) that avoids buffer overflows,
3. and a control mechanism that rejects new input tuples when they lead to an overload.

## 1.3 Objectives

The main objective of this thesis is to address the design issues mentioned above and evaluate a hardware implementation of handshake join architecture. The hardware resource usage and the signal delays are significant factors for the overall design. This thesis also intends to clarify these issues. For this purpose, the proposed design is implemented on an FPGA, and evaluated as a case study in terms of the hardware resource usage, the maximum clock frequency, and the throughput performance.

In our point of view, the major contribution of the thesis is to identify the problems encountered in the design of handshake join hardware. To the best of our knowledge, this is the first work that proposes a complete design of handshake join, and implements it as a dedicated hardware on an FPGA device. The thesis also indicates buffer tuning for join and merge units included in the handshake join architecture. Specifically, it presents analysis regarding buffer-size optimization, and discusses static and adaptive buffer tuning for the proposed design.

Result collection performed by a *merging network* is a significant issue for a handshake join operator. Results from our preliminary evaluation show that the merging network has a potential to be an overwhelming bottleneck for overall performance. It is a crucial limiting factor for the design of handshake join hardware because the throughput performance mainly depends on it, especially at high output rates. The problem is, therefore, how to design and implement an efficient merging network in order to overcome the degradation of performance.

Another objective of the thesis is to address the above problem by proposing an adaptive merging network for the handshake join operator. An appropriate network model and a careful implementation are extremely important to improve performance of the handshake join operator. Accordingly, a markedly different network structure is proposed, overcoming a critical disadvantage of the naively implemented merging network. The handshake join operator with the adaptive merging network substantially outperforms conventional approach. The thesis presents evaluation results of throughput performance compared with another implementation of window-based stream join operator presented in [5].

The rest of the thesis is organized as follows: Chapter 2 briefly reviews previous work. Chapter 3 introduces handshake join and identifies the design issues of handshake join hardware on an FPGA. Chapter 4 presents the details of the implementation of the handshake join architecture, and evaluates the hardware implementation. After that Chapter 5 gives some discussions on buffer size tuning, particularly static and adaptive tuning of the buffers for join and merge units included in the handshake join architecture. Chapter 6 presents the design and implementation of the handshake join hardware with the adaptive merging network, and compares it with the implementation presented in Chapter 4. Finally, Chapter 7 gives conclusions and identifies future work.

# Chapter 2

# Related Work

Chapter 2 briefly reviews related work and provides some background information.

## 2.1 Data Processing on FPGA

Due to increasing demand for processing data streams, DBMS researchers have expanded the data processing paradigm from the traditional store and then process model towards the stream-oriented processing model. An extensive range of research is conducted for new problems owing to the nature of data streams.

The modern CPU architectures are subjected to crucial restriction and limitations. For example, high latency occurs when getting data in and out of the system, and memory wall causes a serious bottleneck in the entire system. Instead of the CPUs, FPGAs can be proposed as an alternative platform to implement data processing systems. In fact, FPGAs are considered as a possible solution for the inherent limitations of classical CPU-based system architectures.

It is shown in [6] that FPGAs are a viable solution for data processing tasks. For example, Sadoghi *et al.* present an efficient event processing platform called *fpga-ToPSS*, which is built over FPGAs to achieve line-rate processing [7]. They demonstrate high-frequency and low-latency algorithmic trading solutions based on the event processing platform [8]. It is stated in [8] that the FPGA-based solution provides a superior end-to-end system performance by eliminating the operating system. They also focus on a multi-query stream processing to accelerate the execution of SPJ (Select-Project-Join) queries [9]. There are other works where FPGA is used as a platform for building application-specific hardware [10, 11, 12].

## 2.2 Window Join Implementation on FPGA

### 2.2.1 Window Join

As mentioned briefly in Chapter 1, how to process joins over unbounded streams is highly problematic. The main reason for why joins are difficult to deal with is that an important assumption made in traditional query processing in DBMSs is no longer valid for stream databases. Traditional DBMSs mainly focus on only fixed-size stored data for query processing. In other words, the DBMSs need to process a *finite* amount of data while executing queries. In addition, the DBMSs have enough time to complete queries including join operations because there is no strict time restriction as in stream databases.

On the other hand, stream databases should process unbounded and continuous streams of data. Contrary to the traditional DBMSs, streams databases require to handle a potentially *infinite* volume

of data while executing continuous queries. Moreover, stream databases are mainly used for real-time applications where low-latency and high-throughput processing are highly essential. Stream databases should meet the strict time restrictions, and this leads to an additional difficulty for stream databases to execute continuous queries including join operations.

In relational databases, each datum is stored as structured data item, called *tuple*, in form of a table. In general, traditional DBMSs may have many tables in their databases each of which includes a finite number of tuples. Basically, a join operator of a traditional DBMS compares each tuple of a table with all tuples of another table in the relational database. In each comparison, a pre-defined condition (*join predicate*) of the join query is evaluated for each pair of tuples. According to the result of the evaluation of the join predicate, the join operator determines whether or not to produce a result tuple of the join query. If the join predicate is satisfied (or the join condition is true), then the compared pair of tuples are combined into a single tuple as a result tuple in the output.

It should be emphasized that the number of combination of tuples is also finite since each table contains a finite number of tuples. For instance, let us say that one table includes $N$ tuples and another table includes $M$ tuples. In this case, the total number of the combination of tuples equals to $N \times M$. When it comes to "joining" these two tables in a join query, the join operator should take into account $N \times M$ combinations of tuples. This is the case for the traditional DBSMs, but not for stream databases.

As mentioned before, stream databases deal with unbounded and continuous input streams. Theoretically, there is no notion of "end of stream" in a data stream. In other words, the input streams are regarded as potentially infinite sequences of input tuples. This leads to a fundamental question that is how to process joins over infinite input streams. It is obvious that we can't handle every combination of tuples from infinite input streams in practice because, as stated in [2], processing a join over unbounded input streams requires unbounded memory.

What causes the problem for processing joins in stream databases is the infinite nature of input streams. Processing all pairs of the tuples from infinite input streams is impossible for a join operator because there is no way to handle the infinite number of combinations of tuples with limited resources. It is necessary to limit the number of tuples that are processed for join operation over the infinite input streams. In order to address the problem, the concept of *windows* is introduced in stream databases for practical applications. In fact, Kang *et al.* mentioned that in practice most join queries over unbounded input streams would contain "window predicates" that restrict the number of tuples that must be stored for each stream [2].

Window-based stream join, which is also called *window join*, can be informally described as follows: A window join operator takes two streams (let us say stream $R$ and stream $S$) as inputs and produces output tuples $\langle r, s \rangle$, where $r$ is from stream $R$ and $s$ is from stream $S$, such that

*1.* $r$ is in the window for stream $R$ at the same time that $s$ is in the window for stream $S$,

   *and*

*2.* $r$ and $s$ satisfy the join condition.

Figure 2.2.1 (adopted from [1]) illustrates an overview of a window join operation over stream $R$ and stream $S$. As shown in Figure 2.2.1, the join operator (indicated as $\bowtie$ in Figure 2.2.1) accepts only finite subsets of the input streams. The finite subsets of data taken from stream $R$ and stream $S$ are indicated as "current window for Stream $R$" and "current window for Stream $S$" in Figure 2.2.1, respectively. In general, the size of each window can differ from each other. In other words, windows over input streams $R$ and $S$ can include the different numbers of tuples. This is illustrated as the different sizes of windows for streams $R$ and $S$ in Figure 2.2.1.

It is stated in [1] that various ways have been proposed to define suitable window boundaries depending on application needs. This thesis focuses on *sliding windows* as in [1] and [5]. Teubner and

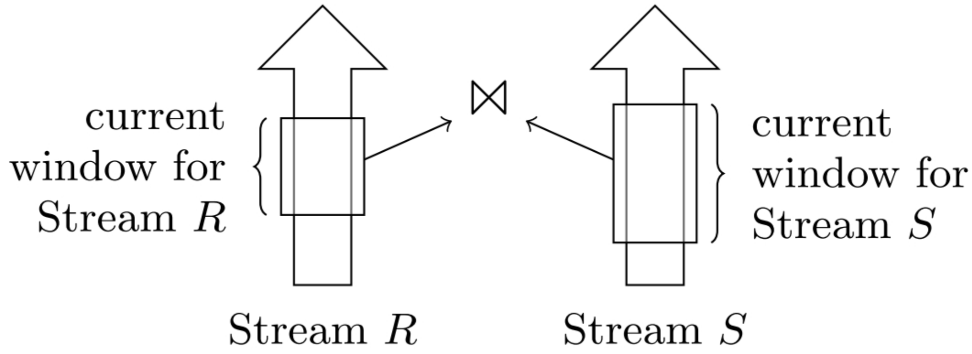



Figure 2.2.1: Window join (figure adopted from [1]).

Mueller [1] describe sliding windows as follows: At any point in time, sliding windows cover all tuples from some earlier point in time up to the most recent tuple. Usually, sliding windows are either time-based, *i.e.*, they cover all tuples within the last $\tau$ time units, or tuple-based, *i.e.*, they cover the last $\omega$ tuples in arrival order.

In accordance with the description of the sliding windows given above, the semantics of window-based stream joins (precisely which tuple coming from an input stream should be paired with other tuples coming from another input stream?) is defined in [1] as follows:

***Semantics of Window-Based Stream Joins.*** *For $r \in R$ and $s \in S$, the tuple $\langle r, s \rangle$ appears in the join result $R \bowtie_p S$ iff ($t_i$ denote tuple arrival timestamps, $T_i$ denote window sizes)*

1. (a) *r arrives after s (i.e., $t_r > t_s$) and s is in the current S-window at the moment when r arrives (i.e., $t_r < t_s + T_S$)*

   *or*

   (b) *r arrives earlier than s (i.e., $t_s > t_r$) and r is still in the R-window when s arrives (i.e., $t_s < t_r + T_R$)*

   *and*

2. *r and s pass the join predicate p (i.e., satisfy the join condition).*

Stream joins based on the sliding windows arise in many practical applications. It is mentioned in [2] that one class of applications in which the sliding windows appear deals with correlating information from different sources about the same entities. For example, we may wish to correlate stock price movements with news stories suspected of influencing the price, or we may want to correlate cell phone traffic with e-mail traffic in a surveillance application [2].

Another class of applications mentioned in [2] deals with tracking entities through a network of sensors. Examples of this sort of application include tracking network packets through routers, or generating “click stream” information about visits to multiple web sites, or even monitoring the progress of cars through tollbooths on the highway [2].

It is essential for some applications dealing with unbounded and continuous streams of data that window join operators should produce the *exact* results of the window join operation. For example, it is probably unacceptable for the window join operator to lost or *drop* some of the result tuples if one is interested in tracking the movements of specific entities [2].

On the other hand, however, there are other kinds of applications that require only a subset of the results, but not the exact results. As an example of this kind of application, Kang *et al.* [2] consider

measuring the delay in traffic between two sensor nodes. They also mentioned that it may be acceptable to compute an average value by looking at a subset of the complete result in the case of such a measurement. In fact, it is indicated in [2] that such an approximate but up-to-date average may be much more desirable than a delayed exact result for real-time applications if the system does not have sufficient resources to produce the complete result in a timely fashion. It has also been called *load shedding* in the stream processing community. There are some studies on good load shedding strategies (*e.g.*, Tatbul *et al.* [13]) that will make the output of stream processing systems most valuable even under high load.

### 2.2.2 Implementation of Window Join

How to implement the join operator is a challenging task in stream databases because of the tight response-time restriction and high input data rates. In addition, stream join operation needs a heavy computational cost. It is necessary to implement an efficient stream join operator in order to overcome these problems. Consequently, acceleration of the stream join operation is a significant research issue regarding stream databases.

Terada *et al.* [5] suggest an implementation technique of window join operator on an FPGA platform in order to improve the performance and achieve high-throughput with low-latency. They try to extract parallelism from the three-step procedure presented by Kang *et al.* [2]. The three-step procedure summarizes operations that a continuous query processor requires to handle when it evaluates a join predicate over windows of input streams.

***The Three-Step Procedure.*** *Let us assume a sliding window join $R \bowtie_p S$ between two input streams $R$ and $S$ where $p$ is a join predicate. Whenever a new tuple $r$ from stream $R$ (i.e., $r \in R$) arrives at the input of the join operator, the three-step procedure is triggered. Each step of the procedure can be described as follows:*

***Step 1:*** *Scan all tuples in the current $S$-window (i.e., $\forall s \in S$-window) to find pairs of tuples $\langle r, s\rangle$ that satisfy the join predicate $p$.*

***Step 2:*** *Insert the new tuple $r$ into the current $R$-window.*

***Step 3:*** *Invalidate all expired tuples in the current $R$-window.*

*A new tuple $s$ arriving from stream $S$ (i.e., $s \in S$) is handled symmetrically.*

It is stated in [2] that even though the steps described above seem simple enough, it turns out that their implementation can become complicated because of a mixture of traditional join processing problems and additional issues introduced by having to evaluate the join using a sliding window over unbounded streams.

Table 2.1: Definition of terms used in cost model

| | |
|---|---|
| $\lambda_r$ | tuple arrival rate of stream $R$ |
| $\lambda_s$ | tuple arrival rate of stream $S$ |
| $W_R$ | current window for stream $R$ |
| $W_S$ | current window for stream $S$ |

Kang *et al.* propose a unit-time-basis cost model for a window-based stream join query [2]. Following the description of the three-step procedure, each tuple arrival in window $R$ triggers three tasks:

*scanning* window $S$ for joining tuples, *inserting* the tuple in window $R$ and *invalidating* any expired tuples from window $R$. Given the notation of Table 2.1, the cost formula for the three-step procedure is shown in Equation 2.2.1.

$$C_{R\bowtie S} = \lambda_r(scan(W_S) + insert(W_R) + invalidate(W_R)) + \lambda_s(scan(W_R) + insert(W_S) + invalidate(W_S)) \tag{2.2.1}$$

The first term of Equation 2.2.1 indicates the processing cost for tuple $r \in R$ arrivals whereas the second term represents the processing cost for tuple $s \in S$ arrivals. In each term, the processing component (*scan, insert, invalidate*) is multiplied by the factor that is the number of tuples arriving per unit time. It should be mentioned that the cost model of the window join operation given as Equation 2.2.1 can be divided into two independent subgroups of operations, one for each input stream. In reference [2], Equation 2.2.1 is rewritten as follows:

$$\begin{aligned} C_{R\bowtie S} &= C_{R\ltimes S} + C_{R\rtimes S} & (2.2.2)\\ C_{R\ltimes S} &= \lambda_s(scan(W_R)) + \lambda_r(insert(W_R) + invalidate(W_R)) & (2.2.3)\\ C_{R\rtimes S} &= \lambda_r(scan(W_S)) + \lambda_s(insert(W_S) + invalidate(W_S)) & (2.2.4) \end{aligned}$$

According to Kang *et al.* [2], rewriting the cost formula (Equation 2.2.1) leads to two important observations:

1. The window join operation is divided into two subcomponents, $R \ltimes S$ and $R \rtimes S$. These subcomponents are called *join directions*.

2. Each subcomponent can be evaluated independently from each other. The cost expression for $C_{R\ltimes S}$ (Equation 2.2.3) is independent of the cost expression for $C_{R\rtimes S}$ (Equation 2.2.4).

Equation 2.2.2 represents the aggregate cost of accessing the windows of each input stream in a single time unit. In particular, Equation 2.2.3 equals to the aggregate cost of accessing $W_R$ in unit time. Similarly, Equation 2.2.4 equals to the aggregate cost of accessing $W_S$ in unit time.

For example, let us think about $C_{R\rtimes S}$ direction (Equation 2.2.4). In a given unit time, $\lambda_r$ tuples arrive from stream $R$ and each of the arriving tuples should be compared with all tuples in $W_S$. This is indicated by the first term $\lambda_r(scan(W_S))$ in Equation 2.2.4. At the same unit time, $\lambda_s$ tuples arrive from stream $S$ and each of the arriving tuples should be inserted into $W_S$. Concurrently, expired tuples in $W_S$ should be invalidated. If we assume tuple-based window, one tuple gets expired for each tuple insertion into $W_S$. The cost of insertion and invalidation are indicated by the second term $\lambda_s(insert(W_S) + invalidate(W_S))$ in Equation 2.2.4.

In practice, Equation 2.2.2, Equation 2.2.3 and Equation 2.2.4 mean that both of the $R \ltimes S$ direction and the $R \rtimes S$ direction can be evaluated in parallel. In fact, the window join operator presented by Terada *et al.* [5] includes two join units one of which is assigned for the $R \ltimes S$ direction and the other is assigned for the $R \rtimes S$ direction. Each of the join units performs the nested loops-style join which is a straightforward implementation of the three-step procedure. As a result, only two join processes are concurrently executed since the approach adopted in [5] is mainly based on sequential execution.

Another implementation of window join on an FPGA device is the M3Join proposed by Qian *et al.* [14]. It is mentioned in [1] that the M3Join implements the join step as a single parallel lookup; however, this approach causes the significant performance drop for larger join windows. On the other hand, the pipelining approach and the data flow model of handshake join do not suffer from these limitations. Details of the handshake join are discussed in the following chapter.

## 2.3 Other works

Handshake join hardware includes join units, namely *join cores*, one of which performs join operation in an independent manner. Join cores only require local core-to-core communication for data transferring. They concurrently perform the same task in a synchronous manner. From this point of view, join cores can be regarded as a one-dimensional *systolic array*.

Kung and Leiserson [15] proposed the idea of systolic array that is a structure composed of an array of processors for VLSI implementation. It is stated in [15] that processing units of a systolic array rhythmically compute and pass data through the system. The data processing and communication model of join cores are consistent with the properties of systolic arrays. In fact, the data flow model of the handshake join is similar to that of the join array [16] proposed for relational databases (*i.e.*, traditional DBMSs). It should be noted that the join array [16] is an implementation of traditional join operation for relational DBMSs. On the other hand, handshake join [1] is an efficient algorithm of window-based stream join operation for DSMSs. This is the difference between the join array and handshake join.

# Chapter 3

# Design of Handshake Join

Chapter 3 introduces handshake join and identifies the design issues of handshake join hardware on an FPGA. In addition, this chapter gives an overview of the proposed design of the handshake join architecture.

## 3.1 Handshake Join

The basic idea of the handshake join [1] is to consider two input streams which are allowed to flow in opposite direction as shown in Figure 3.1.1. With this approach, we obtain significant advantages regarding parallelization and scalability. It is stated in [1] that the parallel evaluation of the matching processes become possible because the approach adopted in handshake join converts the original *control flow* problem (or its procedural three-step description) into a *data flow* representation. It is also stated that there is no hot spot that could become a bottleneck if handshake join is scaled up [1].

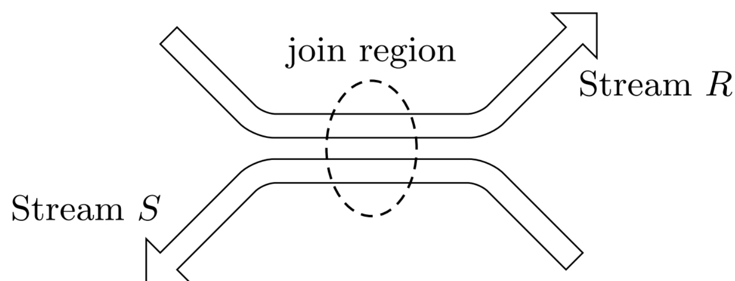



Figure 3.1.1: The basic idea of handshake join (adopted from [1]).

It is mentioned in [1] that, in general, the three-step procedure corresponds to a *nested loops-style* join evaluation; however, the nature of the nested loops-style join evaluation makes it difficult to scale up to a large numbers of processing units. In fact, this is the main reason why only two join processes are executed in [5]. To solve the problem, the distributed data flow-style processing model without a dedicated centralized coordinator is proposed with the handshake join approach. It is indicated in [1] that handshake join produces the same output tuples as classical window-based stream join procedure, and it can be regarded as a safe substitute for traditional window join implementations.

Figure 3.1.2 illustrates handshake join operation for tuple-based windows. In Figure 3.1.2, each rectangular box represents a tuple from two input streams. As shown in Figure 3.1.2, tuples from the input streams $R$ and $S$ are allowed to flow in opposite direction. It is stated in [1] that both join windows

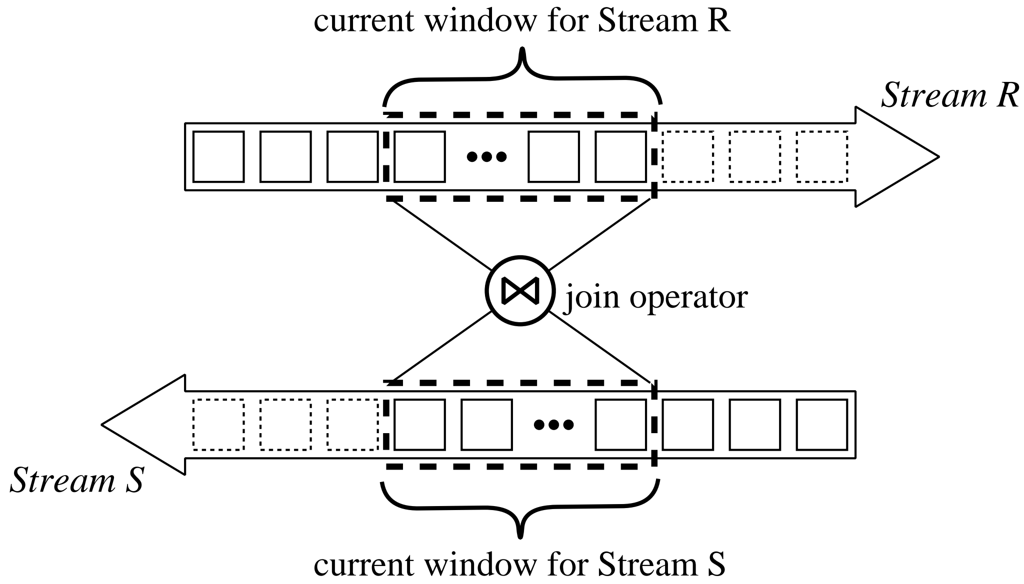



Figure 3.1.2: Handshake join with tuple-based windows. Tuples from the two input streams are flowed in opposite direction.

are lined up next to each other in such a way that window contents are pushed through in opposing directions. It should be also mentioned that the oldest tuple in the current window for stream $R$ or stream $S$ will be expired and discarded whenever a new tuple arrives in the corresponding window. In other words, tuples from the two input streams $R$ and $S$ are pushed through respective join windows, and each tuple pushes all existing window content one step to the side, such that always the oldest tuple "falls out" from the window and expires [1].

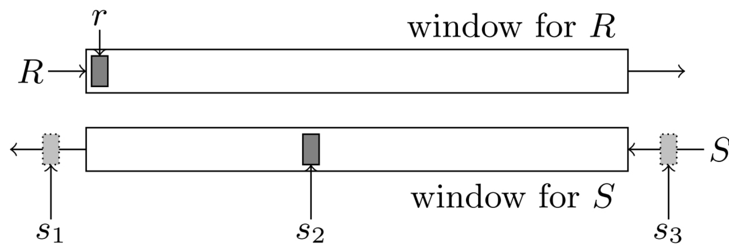



Figure 3.1.3: The semantics of handshake join (adopted from [1]).

Semantics of the handshake join is described in [1] as follows:
***Semantics of Handshake Join.*** Let us assume the handshake join $R \bowtie_p S$ between two input streams $R$ and $S$ where $p$ is a join predicate, and consider the situation at moment $t_r$ when a newly arrived tuple $r$ from input stream $R$ (*i.e.*, $r \in R$) enters its join window as illustrated in Figure 3.1.3. At this moment, a tuple $s_i$ from stream $S$ (*i.e.*, $s_i \in S$) can relate to $r$ in one of the following three situations as indicated in Figure 3.1.3 ($t_i$ denote tuple arrival timestamps, $T_i$ denote window sizes):

*(1)* Tuple $s_1$ is so old that it has already left the current window for stream $S$ (*i.e.*, $t_r > t_{s_1} + T_S$). Thus, $r$ will not see $s_1$ and no attempt will be made to join $r$ and $s_1$. This means that the join predicate $p$ is never evaluated for the tuple pair $\langle r, s_1 \rangle$.

*(2)* Tuple $s_2$ is somewhere in the current window for stream $S$ when $r$ enters the join window for stream $R$. In this case, $s_2$ is older than $r$ (*i.e.*, $t_r > t_{s_2}$), but still within the join window for stream $S$ (*i.e.*, $t_r < t_{s_2} + T_S$).

The two tuples $r$ and $s_2$ are guaranteed to meet eventually since each of $r$ and $s_2$ moves along the corresponding join windows, respectively. In other words, $r$ and $s_2$ move toward each other, and $r$ will see $s_2$ before both $r$ and $s_2$ fall out from the join windows. The tuple pair $\langle r, s_2 \rangle$ will be added to the join result if they pass the join predicate $p$.

*(3)* Tuple $s_3$ has not arrived in the join window for stream $S$ yet (*i.e.*, $t_r < t_{s_3}$). Whether or not an attempt will be made to join the tuple pair $\langle r, s_3 \rangle$ depends on $t_r$, $t_{s_3}$ and the window specification for stream $R$ (*i.e.*, $T_R$). Once $s_3$ arrives, these factors will determine whether $r$ takes a role that is symmetric to cases *(1)* or *(2)* above.

A new tuple $r \in R$ arrives after $s \in S$ in both of the case *(1)* and case *(2)*. In addition, the join predicate $p$ is evaluated for the tuple pair $\langle r, s \rangle$ only when $t_r < t_s + T_S$. From this point of view, both of the cases *(1)* and *(2)* are consistent with the part *1.(a)* of the classical definition of window-based stream join semantics (Section 2.2.1). On the other hand, it is stated in [1] that case *(3)* is the situation where $r$ arrives earlier than $s$, and this case yields the same output tuples as covered by the part *1.(b)* in Section 2.2.1.

This explains that the semantics of handshake join coincides with the classical definition of the window-based stream joins (Section 2.2.1). This also means that handshake join produces the exact same output tuples that the classical three-step procedure does [1]. That's why the approach of handshake join can be regarded as a safe substitute for window join algorithm, and thus, it can be used for implementing window-based stream join operators.

One thing should be mentioned here is that handshake join may produce result tuples in a different order compared with the classical three-step procedure. It is stated in [1], however, that a certain degree of local disorder is already prevalent in real applications. For example, some stream processing engines, such as *Truviso Continuous Analytics* system [17], implement order-independent processing to handle the out-of-order tuples. In fact, it is mentioned in [1] that the different tuple order can be corrected with standard techniques such as *punctuations* [18]. While addressing the problem of the different tuple order that handshake join can produce is an important topic for future work, the issue is out of scope for the purpose of this thesis.

The parallelization of the handshake join operation is illustrated in Figure 3.1.4. As shown in Figure 3.1.4, the degree of parallelism can be easily increased to a higher level than ever achieved before, by increasing the number of processing units (cores). Since each core is responsible for only its own segment of the two stream windows, all tuple comparisons and evaluation of the join condition are carried out locally and independently. Theoretically, it can be readily scaled up in order to support large window sizes, achieve high throughput rates, and/or handle compute intensive functions of the join conditions.

## 3.2 Design Issues of Handshake Join Hardware

Figure 3.2.1 (adopted from [1]) illustrates the general overview of the handshake join with tuple-based window. As shown in Figure 3.2.1, join cores are aligned side by side so that tuples of the stream $R$ and $S$ flow in opposite direction. It can be easily noticed that the windows of the two input streams are divided into $n$ sub-windows over $n$ join cores. Furthermore, FIFO buffers (indicated as ▤ in Figure 3.2.1) are included in each of the join cores and mergers. Three design issues have to be considered in order to implement handshake join hardware based on Figure 3.2.1.

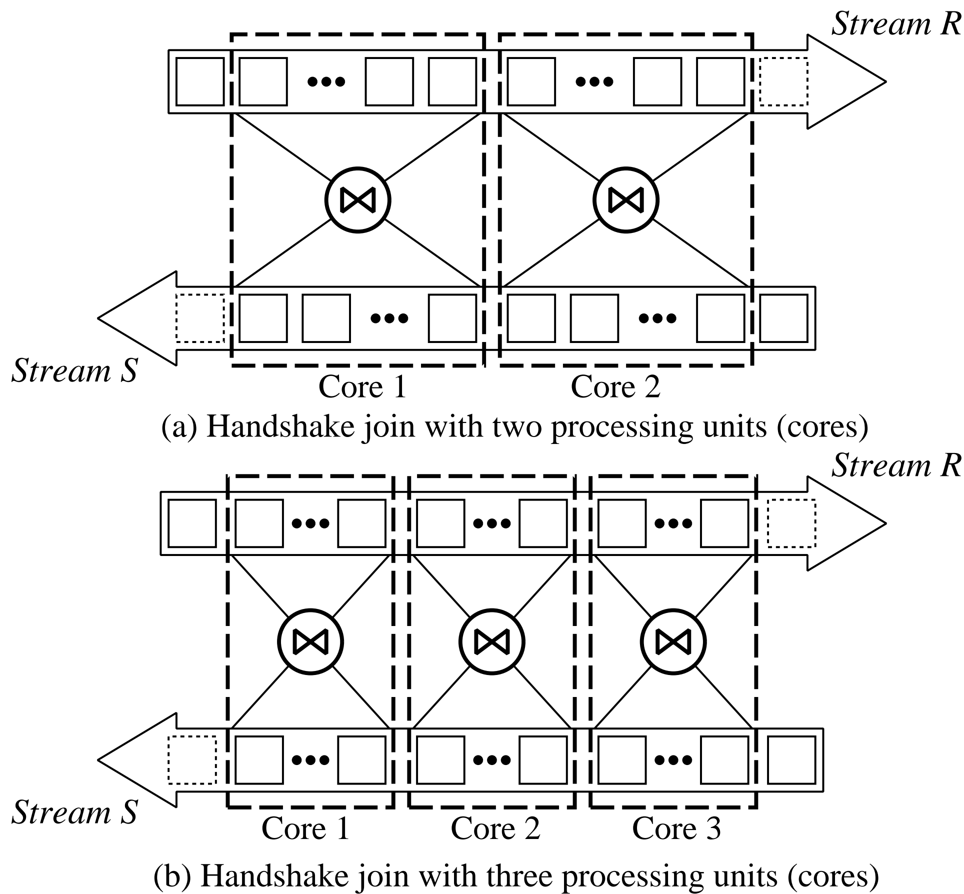



Figure 3.1.4: Parallelization of handshake join operation. Each of the cores evaluates its own segment of the both windows.

First, result collection is a main design issue for handshake join hardware. As illustrated in Figure 3.2.1, the result merging logic is placed on top of the join cores. It is not implemented in [1] even though it is stated that a merging network should merge all sub-results generated by each join core.

The second issue is the limitation of the *bandwidth* of the output channel (bandwidth refers to the amount of data transferred per unit time). There is a possibility that output rates exceed the bandwidth of the output channel, depending on the characteristics of the input streams. It is important for handshake join hardware to be prepared to handle such cases.

Finally, the limitation of the size of the FIFO buffers is considered as a critical issue. Even if most of the meaningful queries would produce a small amount of results, a possibility of buffer overflow still remains in some applications. For instance, a number of tuples satisfying a join condition can arrive from input streams in *TCP SYN Flood* detection [4]. In fact, it depends on the dynamic characteristics of input streams, particularly whether or not a *TCP SYN Flood* attack [19] occurs. This causes an instantaneous overload of the merging network which leads to the risk of buffer overflow. In this case, some of the results overflow out of the buffers and they are permanently lost. Whether or not the problem would occur really depends on application parameters (*e.g.*, input data rate, match rate, and window size); however, handshake join hardware should be prepared to avoid overflow of the FIFO buffers.

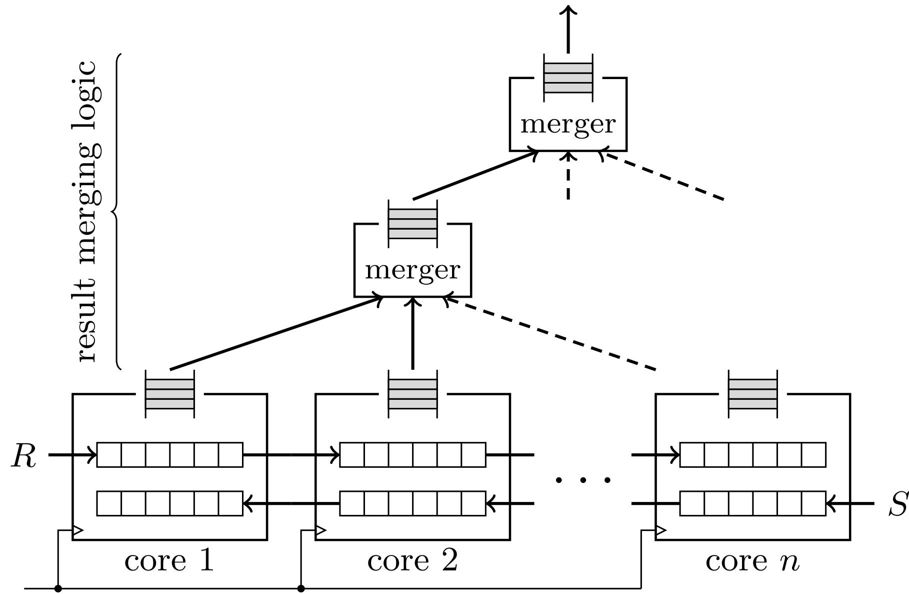



Figure 3.2.1: Overview of the handshake join architecture with tuple-based windows for FPGA implementation (adopted from [1]).

## 3.3 Design Strategy of Handshake Join Hardware

The following components are introduced in the design:

1. join core,
2. merger,
3. merging network,
4. and admission control.

Join cores and mergers are shown in Figure 3.2.1. These are fundamental components for join operation and merging results. Merging network is a result merging logic consisting of a number of merger units. It should be scalable to merge result tuples even if the number of join cores is increased. Admission control mechanism provides a flow control between join cores and the output port to prevent data loss due to the buffer overflow. Moreover, the mechanism rejects input tuples when it is difficult to handle high-rate streams causing an overload. It is designed in a way that the proposed approach can be suitably integrated with a load shedding scheme.

It should be also emphasized that the join cores only require local communication for data transferring, and they are regarded as a one-dimensional systolic array. On the other hand, the proposed design is composed of not only join cores but also the merging network and the admission control mechanism. This is the main difference between a traditional simple systolic array and the proposed design.

# Chapter 4

# Implementation of Handshake Join Architecture

This chapter presents an implementation of the handshake join architecture based on Figure 3.2.1. As stated in Chapter 3, two windows of the input streams $R$ and $S$ are divided into $n$ pieces respectively. Accordingly, each join core is assigned two sub-windows that one comes from stream $R$ and the other comes from stream $S$. Additionally, all of the join cores are connected in such a way that the tuples of the each input stream flow in opposite direction.

Even though it seems that only join cores are driven by a common clock signal in Figure 3.2.1, in the proposed design, merger circuits that compose the result-merging network are also driven by a common clock signal as well as the join cores. Hence, both of the join cores and merger circuits operate synchronously with the same clock signal.

The common clock signal, which is distributed over the whole chip, enables us to design the windows of the each input streams as large shift registers benefitting from the direct support of the FPGA. Consequently, whenever a new tuple arrives, all of the join cores are able to send their oldest tuple to the respective next neighbor simultaneously and thus, an arriving tuple shifts all tuples of the same stream synchronously through the respective window. Actually, because of the data flow model described above, handshake join can accomplish high degree of parallelism without a dedicated centralized coordinator.

As shown in Figure 3.2.1, a hardware implementation of join cores and merger circuits is need to be provided in order to complete the implementation of the handshake join operator. Moreover, although the connection of the join cores is implicitly defined in the handshake join semantic, the architecture of the merging network, which would be composed of merger circuits and their connections, is neither described in the definition of the handshake join algorithm nor illustrated explicitly in Figure 3.2.1. However, how result merging logic should be designed is crucial point for handshake join hardware so as to construct the final output stream by merging all sub-results into a single stream.

So far we have outlined the architecture of handshake join, which is illustrated in Figure 3.2.1, and given the general ideas of the handshake join. Now, let us describe how these circuits are implemented in further detail.

## 4.1 Join Core

The most fundamental circuit in our architecture is, of course, join processor (join core) that evaluates the join condition over the tuples in the windows of the input streams and generates output tuples that compose an output stream. As mentioned before, segments of the windows of the input streams $R$ and $S$ are implemented as large shift registers that hold tuple data of the each stream.

In addition to holding the tuple data, there is a one-bit *valid flag* field for each tuple in the windows indicating whether the respective tuple field is valid or not. That is to say, if a valid flag is set to logic 1, then it means there is a valid tuple datum in respective tuple field, whereas if a valid flag is reset to logic 0, it means the respective tuple field is empty *i.e.*, no valid data.

Besides the large shift registers, which represent the segment of the windows for each input stream $R$ and $S$, there need to be an output buffer that keeps the output tuples generated in the respective join core. For this purpose, a circular FIFO queue is implemented in each join core as an output buffer.

Two types of different embedded memories are available in the Xilinx FPGAs, which are a dedicated Block RAM (BRAM) primitive and a LUT configured as distributed RAM. Our implementation of the FIFO buffer is based on the dedicated BRAM primitive, which is configured as dual-ported RAM, that directly supported by the FPGAs. There might be different use cases of FPGA-embedded memories; however, it is a fact that distributed RAM consumes regular logic cells and hence, it competes for resources with the other circuits, on the other hand, BRAM uses its dedicated resources. Accordingly, we can effectively use our hardware resources available in FPGA devices by utilizing the dedicated BRAM primitives as embedded memory units. It should be also mentioned that we could read from and write to BRAMs one tuple per cycle and in our case, this is suitable for the FIFO buffer implementation.

Furthermore, there are two address registers, which are read-address register and write-address register, inside the FIFO buffer circuit. In addition, two state flags are included in the FIFO buffer, namely empty and full. Although, the registers and the state flags mentioned above may seem self-explanatory, one point should be noticed that full flag is set to logic 1 whenever the FIFO buffer is full or almost full (*i.e.*, there are only few locations left).

The state transition diagram of a join core circuit is illustrated in Figure 4.1.1. The **STATE0** performs a hardware initialization of the join core circuit. The following state, **STATE1**, indicates that a join core is ready for accepting new tuples of the both of the two input streams $R$ and $S$ at the same time. The details of the operations that are carried out in other states illustrated in Figure 4.1.1 are described below.

First, let us look into the two consecutive states that are **STATE2** and **STATE3**. When a new tuple is received from either or both of the input streams, a join core reads its own input ports of the two input streams as well as the respective valid signals that indicate whether the data on the input ports of the tuple field is valid or not. After that, the data read form the each input port is written to input buffer registers respectively with valid flags. At this point, if one of the input tuples has not arrived yet, then respective valid flag will be reset to logic 0 so as to notice that a tuple data written into the corresponding input buffer register is invalid. Otherwise, the valid flags of the input tuples will be set to logic 1. Consequently, input buffer registers and corresponding valid flags have been updated accurately and these buffers are ready to be processed at the beginning of the next state *i.e.*, **STATE4**.

Secondly, after loading the new input tuples in the previous state, that is **STATE3**, there are two possible candidates, which are **STATE5** and **STATE6**, for the next state of the **STATE4** as shown in Figure 4.1.1. The next state will be determined by a condition in **STATE4**. If "valid_R", which represents the valid flag for the most newly arrived input tuple from the input stream $R$, is "False", then it means that the valid flag has been reset to logic 0 and there is no valid data in the input buffer register of stream $R$. Therefore, the segment of the window for stream $R$ will not be shifted. In this case (*i.e.*, when "valid_R" is "False"), we will skip **STATE5** and the next state of the **STATE4** is determined to be **STATE6**. Contrarily, when "valid_R" is "True" that is the valid flag has been set to logic 1, this indicates that there is valid tuple data in in the input buffer register of stream $R$. This time, since a new tuple has come from the input stream $R$, the newly arrived tuple will be inserted in the current window for stream $R$, and thus each segment of the window for stream $R$ has to be shifted one-step to the side. Accordingly, in this case, the next state of the **STATE4** is determined to be **STATE5**.

Thirdly, after a newly received tuple data from the input stream $R$ is loaded into the input buffer register in the previous state, the next step is to insert the received tuple in the current window for the

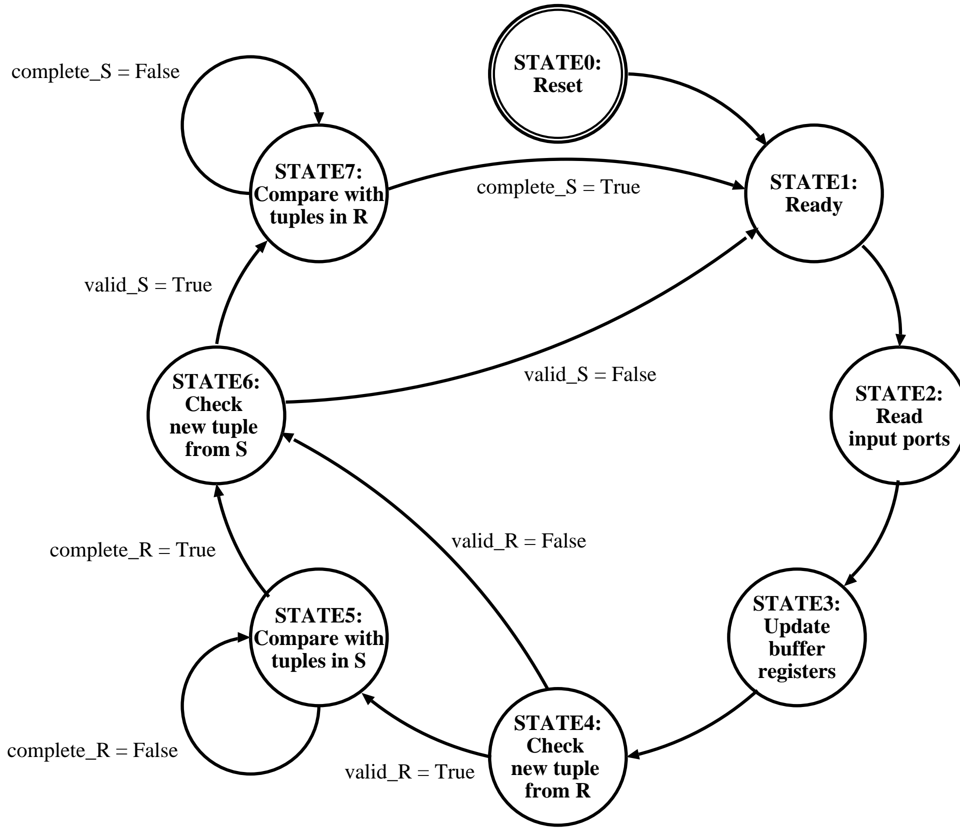


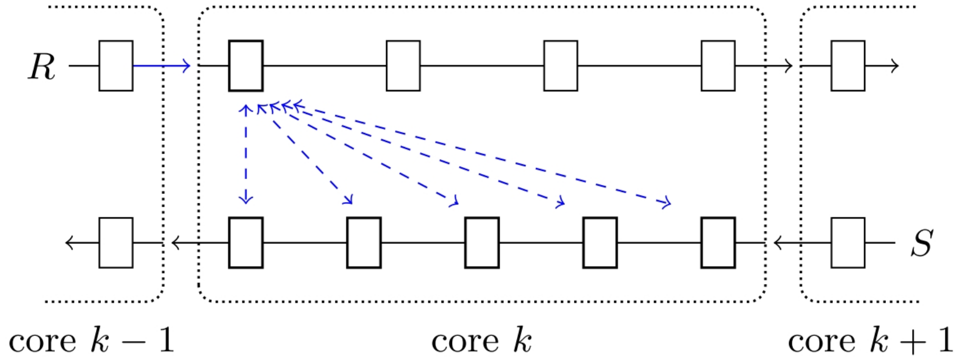

Figure 4.1.1: State transition diagram of the join core circuits.

stream *R*, which is implemented as a large shift register. Consequently, in the **STATE5**, each join core should shift its own segment of the window for stream *R* one-step to the side. At this point, in addition to shifting the window, the key value of the received tuple should be compared with each key value of the tuples in the segment of the window for stream *S* (an equi-join is assumed for simplicity as in [1]). After all, for accurate execution of the window join operation, it has to be guaranteed that the newly received tuple from stream *R* is to be compared with all of the tuples that are in the current window for stream *S*. In order to meet this requirement, we have adopted the immediate scan strategy, which will be described below, that is introduced by Teubner and Mueller in [1]. Hence, we have accomplished whole window semantics correctly by utilizing the immediate scan strategy.

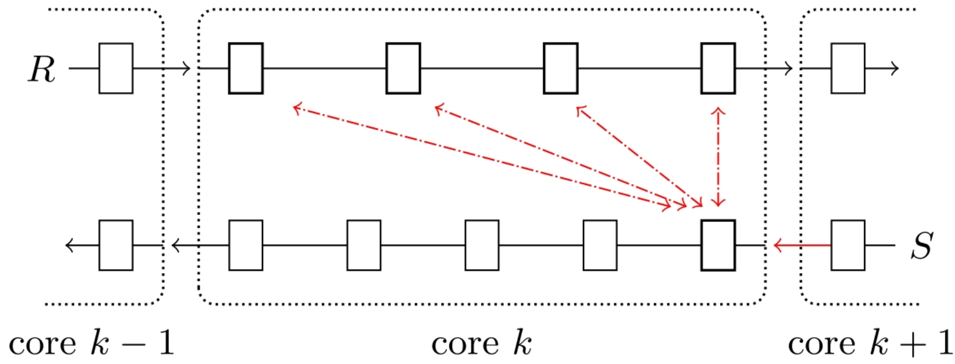
The immediate scan strategy is a local processing strategy that meets the requirement mentioned above, and thus it will guarantee the correct window semantics. In our proposed design, we have used immediate scan strategy with a nested loop join implementation as in [5].

The immediate scan strategy is a specific strategy that is used by each join core so as to execute window join operation on its own segments of the windows for the input streams *R* and *S* . The illustration of the immediate scan strategy for the segment *k* is given in Figure 4.1.2 (adopted from [1]). In this particular illustration, the number of tuples, which are in the corresponding segments of the windows in the join core *k*, differs from each other. As a matter of a fact, the immediate scan strategy can be used in spite of the different window sizes and it works accurately even if the relative window sizes are different from each other.

The immediate scan strategy would work as follows: when a tuple from input stream *R* or *S* enters to



(a) Tuple from stream $R$ entered segment.



(b) Tuple from stream $S$ entered segment.

Figure 4.1.2: Immediate scan strategy. When a tuple from input stream $R$ (a) or $S$ (b) enters to the join core $k$, the immediate comparison will be triggered respectively in the same join core (figure adopted from [1]).

the join core $k$, the immediate comparison will be triggered respectively on the corresponding segments of the windows in the join core $k$ (see Figure 4.1.2). Accordingly, as shown in the illustration given in Figure 4.1.2 (a), after entering the segment $k$ of the window for stream $R$, a newly entered tuple is compared at once with all tuples of stream $S$ that are already in the same segment of the window. Figure 4.1.2 (a) shows all necessary pairs that have to be compared after the tuple $r$ is inserted into the join core $k$. Similarly, when a new tuple from stream $S$ is inserted into the segment $k$, the most recently entered tuple is compared with all tuples of stream $R$ that are already in the same segment of the window as shown in Figure 4.1.2 (b).

Let us come back to the **STATE5** in Figure 4.1.1. As mentioned before, we have adopted the immediate scan strategy with nested loop join in the implementation of our proposed design. That is, all of the necessary comparison mentioned in the description of the strategy is carried out by using the approach of nested loop join. Thus, based on the immediate scan strategy with nested loop join, after inserting the new tuple of stream $R$ into the corresponding window, all comparisons are sequentially performed with the tuples that are in the window of stream $S$ in **STATE5**. Accordingly, during the nested loop join execution, a transition to the next state should not be allowed and the state has to remain at the same state *i.e.*, **STATE5**. In Figure 4.1.1, "complete_R" represents whether the execution of the nested loop join is completed or not. If "complete_R" is "False", then it means that the nested loop join is being

executed, and therefore the state remains **STATE5**. On the other hand, if "complete_R" is "True", then the transition to the **STATE6** is allowed as the nested loop join has already been completed.

Finally, the tasks that should be performed in the remaining states, which are **STATE6** and **STATE7**, are similar to what has been performed in **STATE4** and **STATE5** respectively. The main difference is that **STATE4** and **STATE5** states focus on a new tuple that comes from stream *R*, whereas **STATE6** and **STATE7** states deal with a newly arrived tuple from stream *S*.

## 4.2 Merger

In our proposed design, we have tried to keep the merger circuits as simple as possible. Accordingly, we have designed two-in one-out merger that can be considered as the simplest case, which is slightly different from what is illustrated in the top half of Figure 3.2.1. That is, the only task is to merge two input streams of data into one.

The components included in a merger circuit are very simple: a circular FIFO queue, two input buffer registers and corresponding flags that indicate whether the data contained in each buffer register is valid or not. In addition, it should be also mentioned that the circular FIFO queue is used as an output buffer that keeps the output tuples generated by join cores.

The state transition diagram of a merger circuit is illustrated in Figure 4.2.1. **STATE0** represents the reset state where necessary hardware initialization operations take place. The details of the other states illustrated in Figure 4.2.1 are described below.

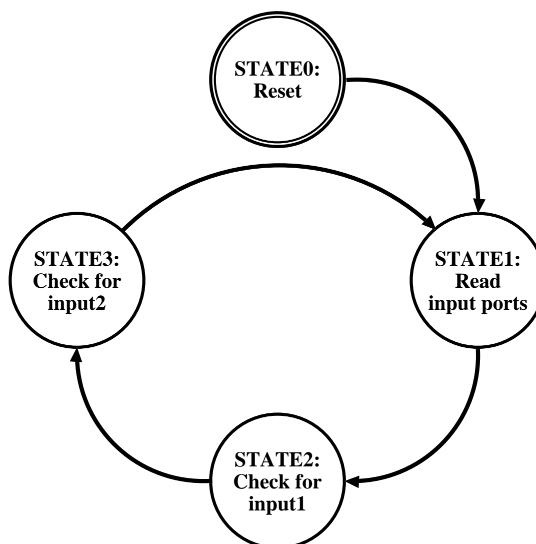



Figure 4.2.1: State transition diagram of the merger circuits.

First, the result tuples, which are generated by join cores, are read form the two input ports and written to the input buffer registers in **STATE1**. At this point, if there is no valid data on one or both of the input ports, then respective valid flag will be reset to logic 0 so as to notice that the data written into the corresponding buffer register is invalid. Otherwise, the valid flags of the data on the input ports will be set to logic 1. Consequently, buffer registers and corresponding valid flags have been updated correctly and these buffers are ready to be used for the next state *i.e.*, **STATE2**.

Secondly, in **STATE2**, after loading the data from the input ports, let us say port1 and port2, if the valid flag of input port1 is logic 1, then a result tuple that is stored in the buffer register is transferred to

the output buffer *i.e.*, the circular FIFO queue. On the other hand, if the valid flag of input port1 is logic 0, there is nothing to be done but transit to the next state.

Finally, the task that should be performed in **STATE3** is very similar to what has been done in the previous state. That is, the only difference is that **STATE2** focuses on the data comes from input port1, while **STATE3** deals with the data comes from input port2. It should be also noted that the next state of **STATE3** is **STATE1**, and therefore a merger circuit would repeats states from **STATE1** to **STATE3** for ever and ever.

## 4.3 Merging Network

As mentioned before, the connection of each join core, which is implicitly defined in the handshake join semantic, is obvious. That is to say, all of the join cores have to be connected in a way that the tuples of the input streams $R$ and $S$ flow in opposite direction. However, as shown in Figure 3.2.1, the architecture of the result merging logic is not given in detail. Furthermore, there is no description about the architecture of the merging network in the definition of the handshake join algorithm.

It is a fact that a result-merging network is needed in order to merge all partial results produced by a number of join cores into a single stream as the final join output. In our proposed design, we suggest a binary tree-like connection for the architecture of result merging network that consists of several merger circuits and their respective connections.

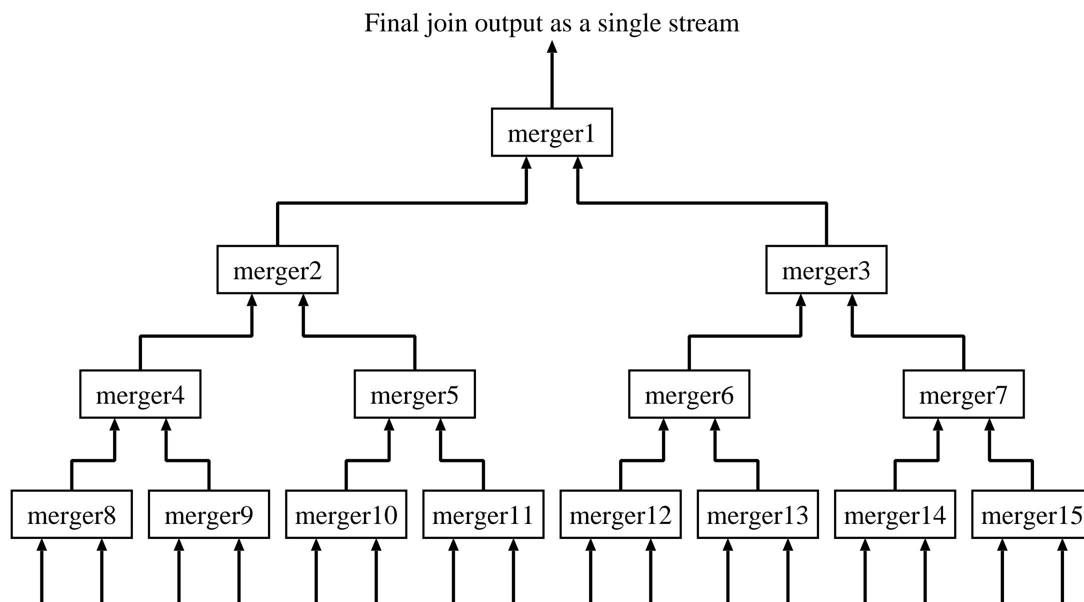



Figure 4.3.1: Binary tree-like connection. An example of a result-merging network for 16 join cores.

An example of a result-merging network for 16 join cores is illustrated in Figure 4.3.1. As described before, we have implemented two-in one-out merger circuit in order to merge two input streams of result tuples generated by join cores into a single output stream. Accordingly, we can use our merger circuits so as to make binary tree-like connections as illustrated in Figure 4.3.1. As shown at the top of Figure 4.3.1, we can obtain the final result of the window join operation as a single output stream from output port of the root node (*i.e.*, *merger1*). Moreover, it should be also indicated that, there are 16 open input ports of which mergers that are numbered from 8 to 15 at the bottom of Figure 4.3.1. Output ports of 16 join core

circuits can be connected to these input ports so that the result-merging network can merge 16 streams of result tuples into a single output stream.

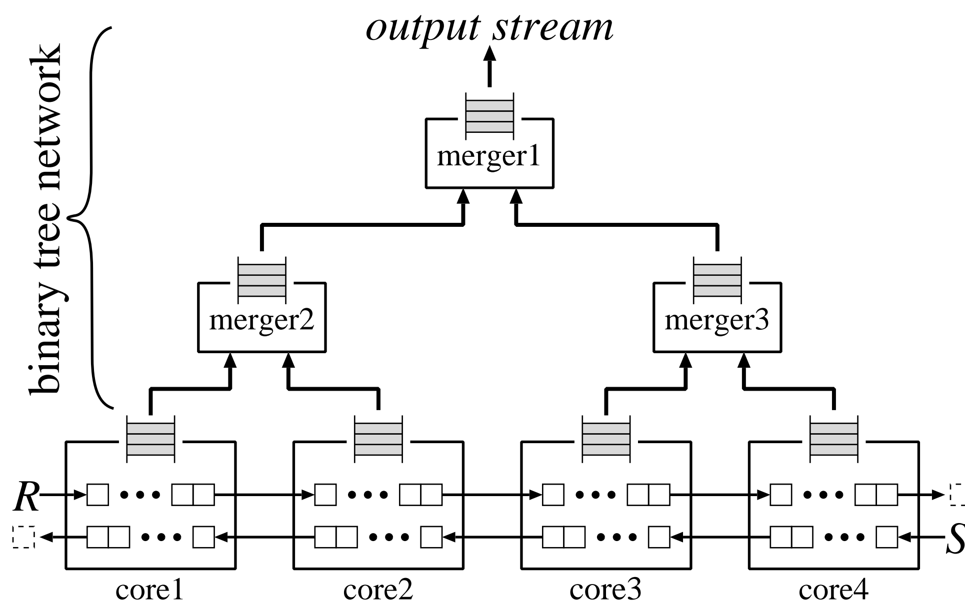



Figure 4.3.2: Connection between join cores and merging network.

For the purpose of clarification, Figure 4.3.2 demonstrates how to connect join cores with corresponding result-merging network. For simplicity, there are only four join cores in Figure 4.3.2; however, the number of join cores and the size of the result-merging network will not affect the approach adopted in Figure 4.3.2.

## 4.4 Admission Control

Admission control mechanism addresses the problem regarding the limitation of the bandwidth of the output channel and the size of the FIFO buffers. The mechanism avoids buffer overflows leading to loss of the results. All results generated by join cores are transferred to the output port by rejecting newly arrived tuples when the output rate exceeds the bandwidth of the channel and/or any of the buffers is close to overflow.

Each FIFO buffer included in a join core or a merger has a *full flag*. It is asserted when the corresponding buffer is almost full (or completely full). The admission control mechanism is summarized as follows:

1. If a full flag is asserted, newly arrived tuples are rejected, and all join cores are suspended until all of the full flags are *de-asserted* (reset to logic 0).

2. Furthermore, if any full flag of mergers is asserted, input ports of the corresponding merger are disabled until its own full flag is de-asserted again.

The overhead of the admission control is as follows. All full flags are *ANDed* together, and the result is stored in a flip-flop. In addition, the output of the flip-flop is connected to each join core. For example, if the number of join core is four (as shown in Figure 4.3.2), there are four-bit signals from the full flags

of join cores and three-bit signals from the full flags of mergers. A total of seven-bit signals are *ANDed* together, and the result is stored in a one-bit flip-flop. This one-bit of information indicates whether or not all of the full flags are de-asserted. With the one-bit signal connected to four join cores, each of the join cores can determine whether to suspend the matching process.

Notice that the problem regarding the bandwidth of the output channel could be resolved by the admission control. For example, in Figure 4.3.2, the FIFO buffer of the *merger1* becomes full when the bandwidth of the output channel is not enough to transfer all results, and the corresponding full flag is asserted. Consequently, the admission control mechanism takes effect in order to prevent loss of the results due to buffer overflow.

What the admission control provides is the flow control between each join core and the output channel. With the admission control mechanism, the proposed handshake join operator takes responsibility for input tuples accepted by join cores. This means that all results derived from the accepted tuples are transferred to the output channel. In other words, no data loss occurs between each join core and the output channel. On the other hand, this does not always prevent loss of actual join results. The loss of the results can occur when the join operator could not keep up with a high input data rate. It is the fact that a lossless flow of all join results is impossible in such cases since some of the input tuples would be rejected (because of the admission policy). It is stated, however, in [1] that load shedding [13] or distribution [20] can be used if handshake join alone is not sufficient to sustain load. The admission control is consistent with load shedding techniques even though implementation of such a mechanism is out of scope of the thesis. The handshake join operator can produce more valuable results once a load shedding mechanism reduces the load of the system because what the admission control guarantees is the join results of the input tuples accepted by the join operator.

## 4.5 Evaluation of the Handshake Join Hardware

The design is implemented on a Xilinx XC6VLX240T-1 chip (Table 4.1). The FPGA design software used in this work is Xilinx ISE 13.1 Logic Edition.

Table 4.1: Specifications of XC6VLX240T-1

| | |
|---|---:|
| #. of Slice Registers | 301,440 |
| #. of Slice LUTs | 150,720 |
| #. of Slices | 37,680 |
| #. of BRAM (36Kbit) | 416 |
| #. of DSP48 | 768 |

### 4.5.1 Resource Usage and Signal Delay

The hardware resource usage and clock frequency are evaluated for 6 different configurations. The different number of join cores ($2^i$ *where* $i = 1, \ldots, 6$) are instantiated on the FPGA. The parameters used during the instantiation process are as follows. The window size of each join core is set to 8 tuples. Each input tuple consists of 64-bit of data half of which is join key and the remainder is allocated for payload field. A result tuple is composed of 32-bit join key and two payload fields, a total of 96-bit data.

The maximum clock frequency of the prototype system is shown in Figure 4.5.1. The x-axis and the y-axis represent the number of join cores and the clock frequency, respectively. As shown in Figure 4.5.1, the graph is almost constant around 150MHz and the frequency is not declined with increased number of join cores.

The hardware resource usage is given in Table 4.2. In addition, the percentage of the overall resource consumption is shown in Figure 4.5.2. In this graph, the y-axis represents the percentage of the used resources. As shown in Figure 4.5.2, the graph is almost linear, and it can be understood that up to 64 join cores can be instantiated on the FPGA. The results of Figure 4.5.1 and Figure 4.5.2 lead us to the conclusion that the proposed design is scalable in terms of the resource usage and the signal delay.

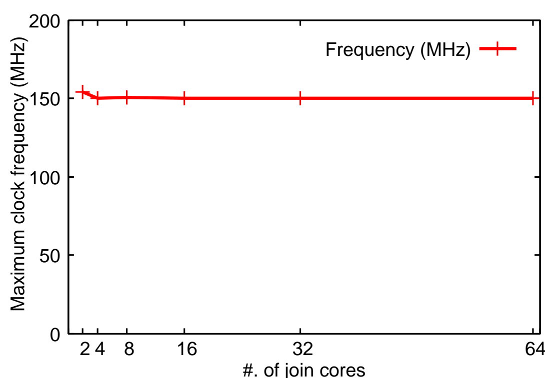



Figure 4.5.1: Maximum clock frequency.

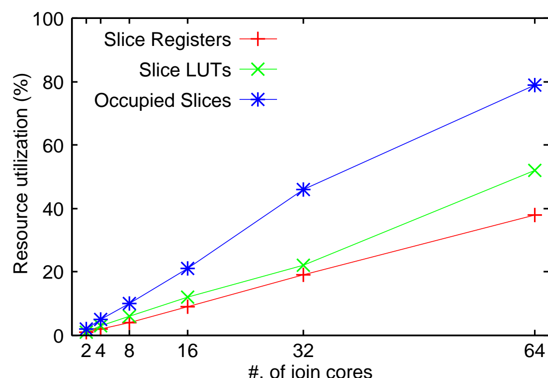



Figure 4.5.2: Overall resource consumption.

Table 4.2: Hardware resource usage

| Join cores | Slice Registers | Slice LUTs | Occupied Slices |
|---|---|---|---|
| 2 | 3,682 | 2,654 | 1,084 |
| 4 | 7,106 | 5,281 | 1,898 |
| 8 | 13,949 | 9,281 | 4,064 |
| 16 | 27,763 | 18,260 | 7,930 |
| 32 | 58,212 | 34,467 | 17,695 |
| 64 | 116,165 | 78,958 | 30,052 |

### 4.5.2 Performance Evaluation

A simple evaluation model can be used as shown in Figure 4.5.3 to evaluate the throughput performance of the architecture. A number of input tuples are generated according to match rates and stored in the input buffer in a random order (according to a uniform distribution). After that, input tuples are transferred to the handshake join operator. While processing the input tuples, it generates result tuples, and they are stored to the output buffer.

The following parameters are used in the evaluation. The handshake join operator includes 64 join cores, and it runs at 100MHz. The size of the input buffer is set to 512 tuples, which is the same as the total size of the window. The sizes of the FIFO buffers included in each join core and each merger are set

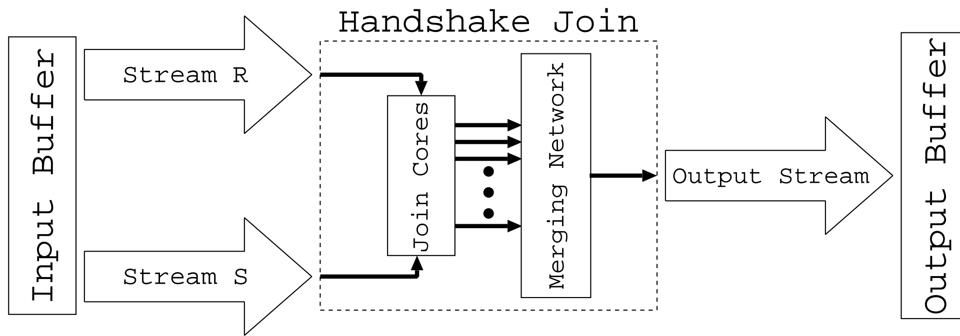



Figure 4.5.3: Evaluation of the handshake join architecture.

to 8 and 4 tuples, respectively. It should be noted that all results generated by join cores are transferred to the output buffer owing to the admission control. This is confirmed by counting the number of results stored in the output buffer. It is shown that the admission control can work properly (no overflow occurs) even if the sizes of the FIFO buffers are set to such a small value.

The throughput is shown in Figure 4.5.4. The line labeled "nested loop join" is the performance estimation of nested loops-style join implemented in [5]. The same parameters as handshake join are used for performance comparison: the size of the input buffer is 512 tuples and it also runs at 100MHz. The y-axis of Figure 4.5.4 represents the maximum throughput of input streams that can be handled by each join operator without dropping any input tuple.

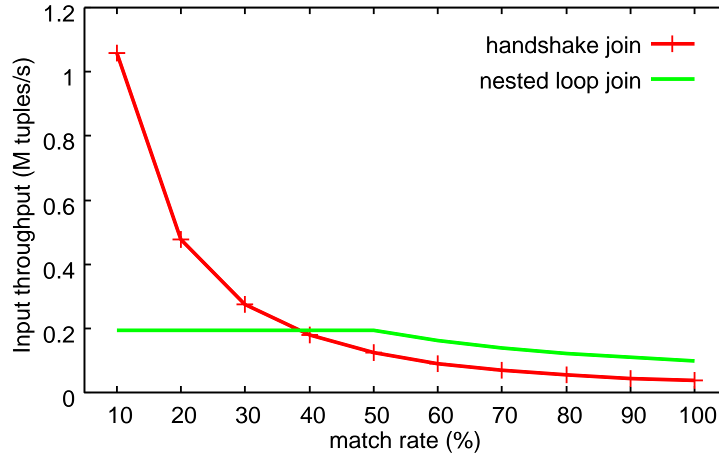



Figure 4.5.4: Maximum input throughput.

Three critical points where an overload can occur should be considered to understand Figure 4.5.4. The first point is between input ports and join cores. As shown in Figure 4.5.3, two input streams (*R* and *S*) flow into the join cores. The second point is between the join cores and the merging network where each join core transfers its sub-results to the merging network. Finally, the third point is between the merging network and the output port. It is the fact that the throughput of the join cores should not depend on match rates. On the other hand, the output rate of the join cores does vary depending on match rates even if the throughput of the input streams remains the same. Furthermore, the parallel execution of joins results in a high output rate because a larger number of results can be produced per unit time

(compared to nested loops-style join evaluation) even though the total number of results is not affected by the execution method. With increasing match rates, the join cores produce a considerable number of results, and therefore the second point tends to overload. In addition, the bandwidth of the output channel strictly limits the throughput of the merging network, and this causes an overload in the third point. In fact, what determines the throughput of the entire system is not the join cores but the merging network, especially at a high match rate. This is because the merging network becomes a critical bottleneck as match rate increases.

On the other hand, low match rates lead to low output rates of the join cores. In such cases, the load of the merging network decreases, and the merging network is no longer the critical bottleneck of the overall system. When a new input tuple arrives in the system, matching processes can be completed in a shorter period of time than the nested loops-style join, taking advantage of the parallel execution of the join cores. That's why the handshake join can achieve higher throughput than the nested loops-style join when match rate is low.

# Chapter 5

# Discussion on Buffer Size Tuning

There is a close relation between the size of the FIFO buffers and the frequency of interruption caused by the admission control. Theoretically, the admission control never suspends the join cores provided that there is enough space in the buffers. On the other hand, limitations of hardware resources should be considered in practice, and allocation of finite buffer space has become an important design issue. It is necessary to clarify how buffer sizes affect the overall performance of the architecture. Chapter 5 gives some discussions on buffer size tuning, particularly static and adaptive tuning of the buffers for join and merge units included in the handshake join architecture.

In order to investigate the effect of the buffer sizes, we use a cycle-accurate simulator of the architecture as a simulation platform. The buffer sizes can be easily modified and this enables us to evaluate the architecture for different buffer size configurations more easily. A huge memory block can be allocated for each buffer of join core or merger by using the software model. As a result, it is also possible to evaluate the architecture in the ideal condition regarding buffer sizes.

The same parameters as in Section 4.5.2 are used in the simulation except for the FIFO buffers. The size of the input buffer is 512 tuples, and there are 64 join cores one of which can store up to 8 tuples for each stream. Input and result tuples are 64-bit and 96-bit wide, respectively.

## 5.1 Static Tuning

As shown in Table 4.1, there are 416 BRAMs each of which can store up to 36Kbit data in a XC6VLX240T-1 chip. That is to say, we can allocate up to $2^{11}$ tuples for each join core and merger when BRAM resources are equally allocated among all of the FIFO buffers which are included in join cores and mergers. From this point of view, the total number of cycles required for completion of the join operation is evaluated for different buffer sizes ($2^i$ *where* $i = 2, \dots, 11$).

The evaluation results are shown in Figure 5.1.1. The simulations are performed at the 100% match rate. The x-axis of Figure 5.1.1 represents the buffer sizes of each node (*i.e.*, join core or merger). These numbers indicate the maximum number of result tuples that can be stored in each FIFO buffer. The y-axis of Figure 5.1.1 stands for the total number of cycles required for completion of the handshake join operation. This means that all of the result tuples generated by each join core are transferred to the output port as a single output stream. Thus, no result tuples remain in any of the FIFO buffers when the operation is completed.

According to the results, the numbers of cycles required for the completion of handshake join are 1464443, 1054029 and 1054012 when the buffer sizes are $2^2$, $2^3$ and $2^4$ respectively. As shown in Figure 5.1.1, results indicate that the total number of cycles is unchanged when the buffer size of each node is equal to or more than $2^4$ tuples.

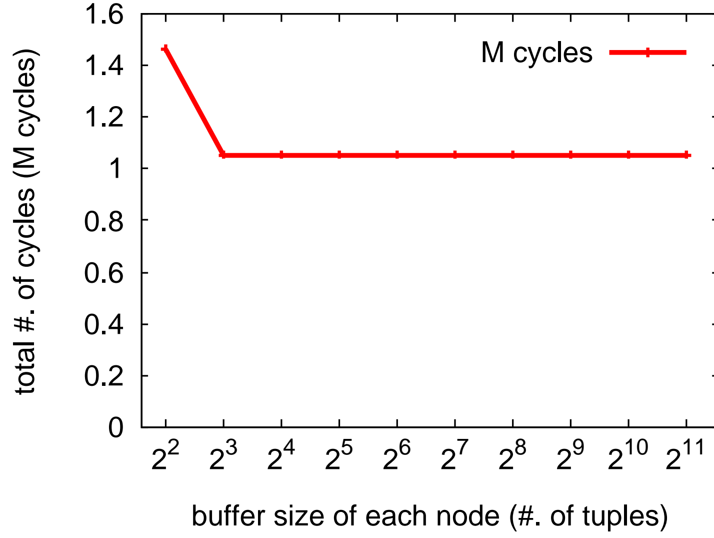



Figure 5.1.1: Results of the simulation for different buffer sizes ($2^i$ *where* $i = 2, \ldots, 11$) at 100% match rate.

What determines the total throughput of the system is not the join cores but the merging network at a high match rate. This is because the merging network becomes a critical bottleneck as match rate increases. Inputs and output of the merging network are critical points, which can become a bottleneck for the overall system performance. The connection point between the join cores and the merging network becomes a major bottleneck when the buffer size of each node is less than $2^4$ tuples. It is possible, however, to alleviate the bottleneck by increasing the buffer size up to $2^4$ tuples. Once it has reached $2^4$ tuples, the main bottleneck is shifted to the output of the merging network since the bandwidth of the output channel strictly limits the throughput of the merging network. As a result, the increased buffer size no longer alleviates the bottleneck; thus, the total number of cycles is constant when the buffer size is equal to or more than $2^4$ tuples.

Table 5.1: Buffer size configurations

| Level of the tree | #. of nodes | config1 | config2 | config3 |
|---|---|---|---|---|
| 0 (root node) | merger × 1 | $2^2$ | $2^{10}$ | $2^8$ |
| 1 (nodes at depth 1) | merger × 2 | $2^2$ | $2^6$ | $2^7$ |
| 2 (nodes at depth 2) | merger × 4 | $2^2$ | $2^5$ | $2^6$ |
| 3 (nodes at depth 3) | merger × 8 | $2^3$ | $2^4$ | $2^5$ |
| 4 (nodes at depth 4) | merger × 16 | $2^4$ | $2^3$ | $2^4$ |
| 5 (nodes at depth 5) | merger × 32 | $2^4$ | $2^2$ | $2^3$ |
| 6 (leaf nodes) | join core × 64 | $2^4$ | $2^2$ | $2^2$ |

So far, the simulation model assumes the same sizes for each buffer of join core and merger. In other words, BRAM resources are uniformly distributed among all FIFO buffers. As the next step, the total number of cycles for non-uniform configuration is evaluated. The details of the three different configurations are given in the Table 5.1. The first column represents level of the tree. Here, the depth of a node is defined as the length of the path from the root to the node. As a special case, the depth of the root node is 0. The set of all nodes at a given depth is called level of the tree. In these configurations, the buffer size of all nodes at the same depth is equal, and each row of the Table 5.1 corresponds to the size of each buffer in the same level.

The total buffer sizes of the each configuration 1, 2, and 3 are 1884, 1920, and 1792 tuples, respectively. Note that the total buffer size is 2032 when the buffer size of each node is equal to $2^4$. We compare the number of cycles required for completion of the operation for these four buffer configurations under different match rates in order to clarify the effect of the difference of the buffer allocation method.

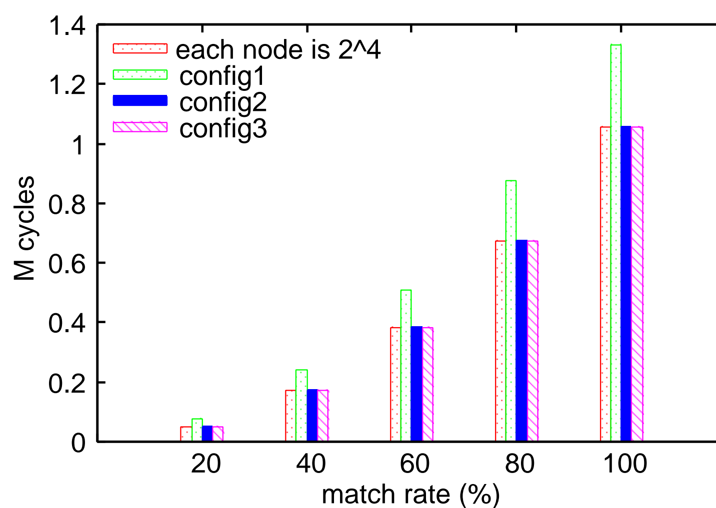



Figure 5.1.2: Results of the simulation for four different configurations at 20%, 40%, 60%, 80% and 100% match rates.

The results of the cycle-accurate simulation are shown in Figure 5.1.2. Results indicate that the buffer allocation methods may have great impact on the performance of the handshake join architecture. It is predictable from these results that the buffer sizes of nodes closer to the root should be relatively larger than other nodes located in deeper levels so as to utilize the limited resources efficiently.

## 5.2 Adaptive Tuning

In the previous section, we focus on the static buffer tuning in order to investigate the relation between the buffer sizes and the performance of the architecture under the condition of limited hardware resources. In this section, we consider the possibility of adaptive buffer tuning for the architecture.

In this evaluation, we assume that the admission control mechanism never interrupts the handshake join operation. Relatively large memory blocks are allocated for buffers of join cores and mergers. In fact, the buffer size of each node is equal to or more than $2^{16}$ tuples. These values guarantee the above mentioned assumption.

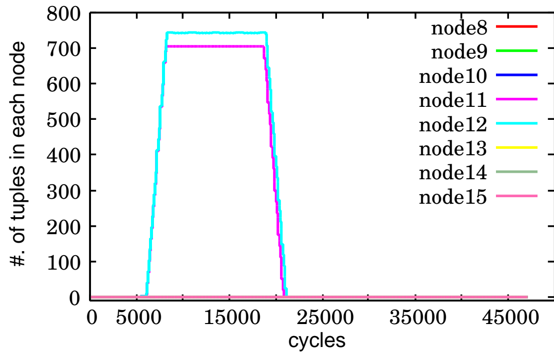
In this simulation, the architecture is evaluated with input streams of three different characteristics so as to investigate the relation between the characteristic of the input streams and the number of tuples

inserted into each buffer. Input tuples which satisfy the join condition are located in the input buffer as follows:

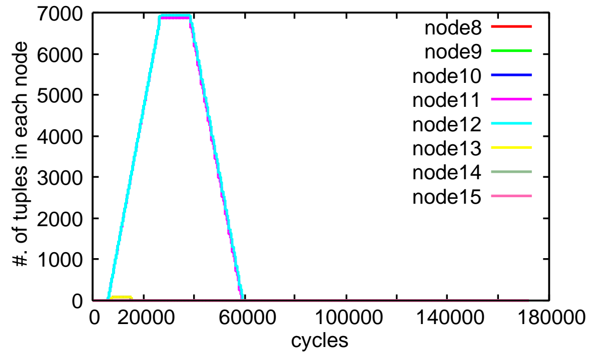
1. according to a uniform distribution,
2. according to a Gaussian distribution,
3. and burst inputs (consecutive tuples that satisfy the join condition).

The results of the cycle-accurate simulation are shown in Figure 5.2.1. In each graph, the x-axis represents the cycles (elapsed time), and the y-axis stands for the number of tuples stored in the buffer at each cycle. Each graph in Figure 5.2.1 corresponds to the nodes at depth 3 in the binary tree (merging network). Results indicate that the number of tuples stored in the buffer differs from each other.

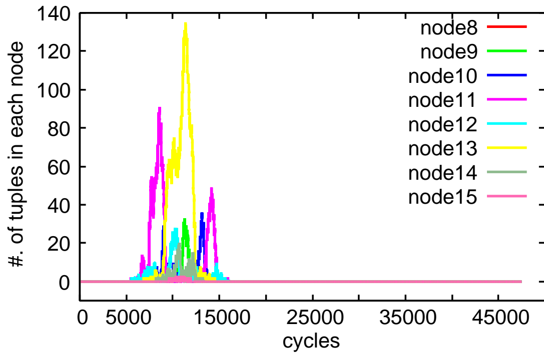
These data lead us to the conclusion that the adaptive buffer tuning can be applied to the architecture because sufficient space is available in some buffers when some of the others store a relatively large number of tuples. These observations imply that some load-balancing methods such as Dynamically Allocated Multi-Queue Buffers [21] can be used for the purpose of adaptive tuning. In this work, an "adaptive merging network" is proposed to address the problem. Details of the adaptive merging network are discussed in the following chapter.



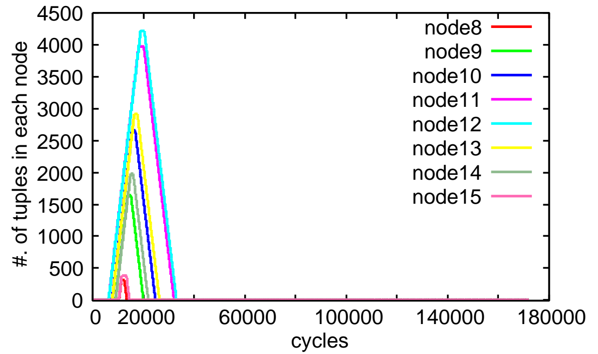
(a) Burst with 20% match rate



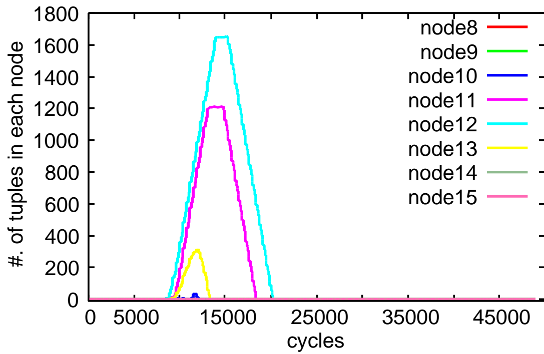
(b) Burst with 40% match rate



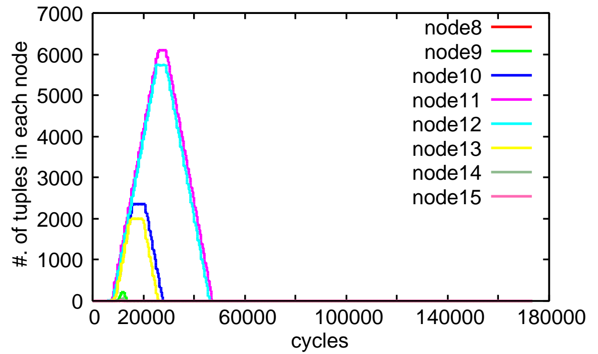
(c) Uniform distribution with 20% match rate



(d) Uniform distribution with 40% match rate



(e) Gaussian distribution with 20% match rate



(f) Gaussian distribution with 40% match rate

Figure 5.2.1: Results of the simulation for input streams of three different characteristics at 20% and 40% match rates.

# Chapter 6

# Design and Implementation of Adaptive Merging Network

The handshake join operator presented in Chapter 4 can achieve high throughput rate compared to [5] when the match rate is low. The merging network, however, suffers from congestion and it becomes a critical bottleneck for the performance of the system if the match rate is increased. Consequently, the performance is considerably degraded when the match rate is high. Chapter 6 proposes an alternative merging network structure, namely *adaptive merging network*, to address the problem. Furthermore, this chapter presents the design and implementation of the handshake join hardware with the adaptive merging network, and compares it with the implementation presented in Chapter 4.

## 6.1 Design Overview

It is indicated in Chapter 4 that the handshake join operator can transfer all results without loss of output tuples by implementing the binary tree network and the admission control mechanism. However, there is a structural disadvantage concerning efficient use of buffers. There is only one path from each join core to the output port because join cores are located at leaf nodes of the binary tree network and all of the result tuples are forwarded towards the root node of the tree. This can cause a problem if the output rate of a join core, which is a measure of how frequently result tuples are generated by a join core, significantly differs from those of others.

In handshake join, each join core evaluates the join condition over the tuples in its sub-windows of input streams. At the same time, result tuples are generated by each join core only if the join condition is satisfied. Whether a join core generates a result tuple or not completely depends on nature of the input tuples being evaluated (*i.e.*, it is data dependent). Accordingly, the output rate of each join core can be time-variant depending on dynamic characteristics of the input streams.

The variation in output rates has to be taken into account when designing architecture of merging network for handshake join hardware even though it is ignored in Chapter 4. For simplicity, let us think about the case that only one join core generates output tuples continuously within a certain period of time. For example, let's say that *core2* in Figure 4.3.2 is the join core that generates outputs. In this case, result tuples are first stored in the buffer of *core2*. After that, they are forwarded to *merger2* and stored in its buffer. Finally, *merger2* transfers result tuples to *merger1*, and they are stored in the buffer of *merger1*.

Although the total number of available FIFO buffers in Figure 4.3.2 is seven, only three of them can be used for buffering results generated by *core2*. This means that more than half of buffers are unusable when the number of join cores is four. Furthermore, the buffer utilization is significantly decreased if

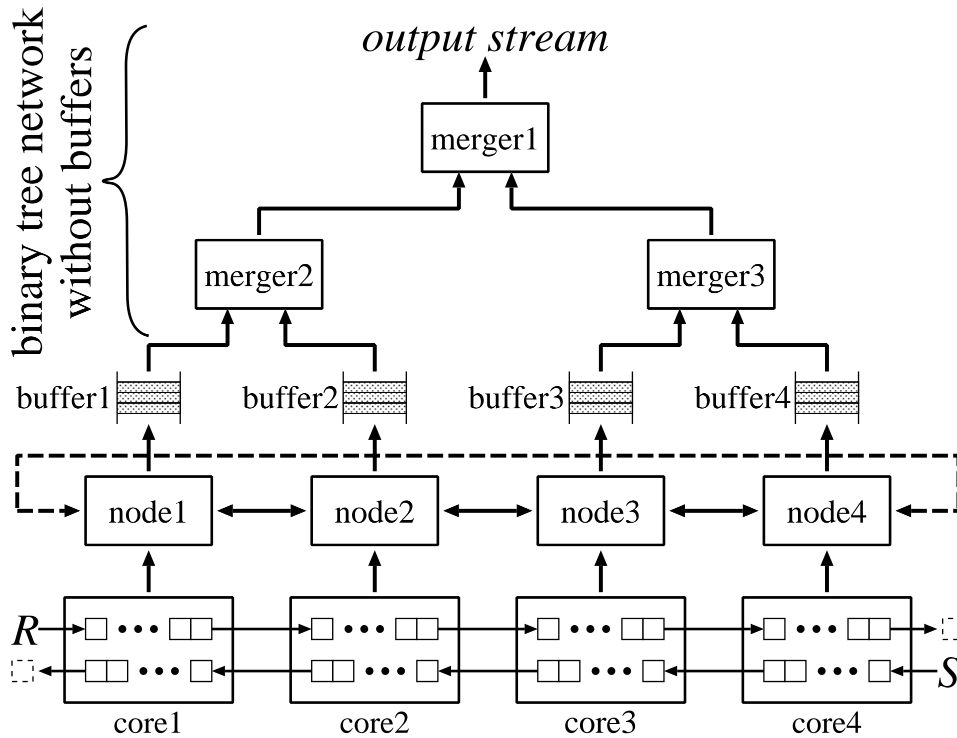



Figure 6.1.1: Adaptive merging network with four bufferless join cores.

the number of join cores is increased. Still, there is no problem provided that the output rate of *core2* remains below the bandwidth of the merging network and the output channel.

However, the admission control mechanism suspends operations of the join cores if the available bandwidth is not enough to transfer all of the results. In this case, new tuples of input streams are rejected and join operations are suspended even though there are unused buffers in overall system. Consequently, the merging network proposed in Chapter 4 has the potential to be a critical bottleneck for the throughput performance.

It can be understood from the fact that the throughput performance of the handshake join operator is strictly limited by the merging network even though it is stated in [1] that handshake join supports high degrees of parallelism and ensures the scalability. The output rate can easily exceed the bandwidth of the merging network since parallelized execution of join operations results in higher output generation rates.

In order to address the problem, an adaptive merging network is proposed as indicated in Figure 6.1.1. The FIFO buffers are omitted from both join cores and mergers. In addition, a new layer of *nodes* and the FIFO buffers are located between join cores and mergers.

As shown in Figure 6.1.1, there are bidirectional links between each adjacent *node*. It should be noted that *node1* and *node4* are also connected by a wraparound link (shown as a broken line in Figure 6.1.1), and therefore, these nodes can be regarded as a ring structure. These links enable two-way data transmission between neighboring nodes in the ring structure. As a result, contrary to the merging network proposed in Chapter 4, each result tuple can take different paths through the merging network to reach the output port of the root node (*merger1* in Figure 6.1.1).

The problem regarding the hardware architecture of handshake join is discussed and an overview of the architecture of the adaptive merging network is given so far. In the following section, an implementation of the proposed architecture is described in more detail, especially the difference between the

handshake join with the adaptive merging network and one proposed in Chapter 4.

## 6.2 Implementation of the Proposed Architecture

### 6.2.1 Join Core

Join cores evaluate the join condition over the tuples in the windows and generate output tuples. Each segment of the windows is implemented as a shift register. The common clock signal, which is distributed over the whole chip, enables us to design the windows of the each input streams as large shift registers benefiting from the underlying FPGA hardware. In addition, there is a one-bit *valid flag* field for each tuple in the windows indicating whether or not the corresponding tuple field is valid.

Whenever a new tuple arrives, all of the join cores send their oldest tuple to the respective adjacent cores simultaneously. Therefore, a newly arrived tuple can shift all tuples of the same stream throughout the window. After that, each join core compares the key value of the received tuple with the key values of all tuples in another segment of the window. Because of the data-flow model described above, join cores require no centralized coordinator that manages overall data-flow among them.

The implementation of a join core is based on Chapter 4; however, the main difference is the existence of the FIFO buffer that stores output tuples generated by the join core. In the proposed design, the join cores forward the result tuples directly to the merging network as shown in Figure 6.1.1 instead of storing them in the buffers.

### 6.2.2 Adaptive Merging Network

The adaptive merging network is the most important and notably different part of the handshake join architecture proposed in this chapter compared to one proposed in Chapter 4. The simple binary tree network only composed of the mergers is proposed in Chapter 4 as the merging network for the handshake join operator. By contrast, a totally different network model is adopted here. The adaptive merging network is composed of the binary tree network (without buffers), a layer of FIFO buffers and the ring structure directly connected to the join cores. With the proposed adaptive merging network, the handshake join operator can accomplish much higher data throughput than ever achieved before (see Section 6.3.3 for more details).

#### Binary tree network

The binary tree network proposed in Chapter 4 (Figure 4.3.2) includes the FIFO buffers and it is responsible for two main tasks:

1. to buffer result tuples coming from each join core,
2. and to generate a single output stream by combining streams of sub-results produced by multiple join cores.

In contrast to the previous merging network, the binary tree network included in the proposed merging network (Figure 6.1.1) has no FIFO buffers and it is no longer responsible for buffering results.

Each merger circuit has two input and one output ports for data transfers. That is, the one and only task is to merge two streams of data into one. The mergers share a common clock signal with join cores. The components included in a merger circuit are very simple: two input buffer registers and an output buffer register with *valid flags* indicating whether or not the data stored in each register is valid.

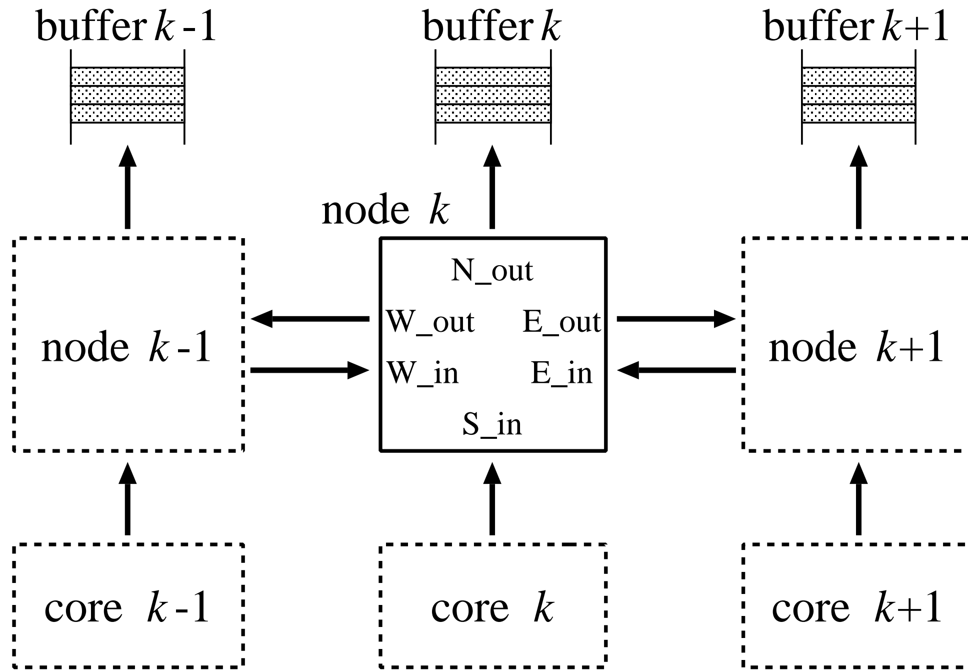



Figure 6.2.1: The connections of the ring structure in the proposed merging network.

The difference between the proposed merger and the one proposed in Chapter 4 is the existence of the FIFO buffer that stores result tuples. As shown in Figure 6.1.1, each merger forwards the result tuples from its two input ports directly to the output port instead of storing them in buffers.

**Ring structure and FIFO buffers**

The connections of the ring structure in the proposed merging network is shown in Figure 6.2.1. In this proposed design, the bidirectional links are considered as two directed links between each pair of nodes. Each node requires two input and two output ports to connect to adjacent nodes. Moreover, additional input and output ports are required to connect to a join core and a FIFO buffer, respectively. Therefore, a total of six ports are available for each node of the ring structure for data transfers. As shown in Figure 6.2.1, *node k* has three input ports (*S_in*, *E_in* and *W_in*) as well as three output ports (*N_out*, *E_out* and *W_out*).

Each node of the ring structure shares a common clock signal with join cores and mergers. It contains a buffer register for each of the output ports, which are *N_out*, *E_out* and *W_out*. These buffer registers are used to transfer the result tuples coming from a join core connected to *S_in*, and adjacent nodes connected to *E_in* and *W_in*. It should be noted that each buffer register can store only one tuple at a time, and there is no FIFO buffer in *node k*.

The proposed design adopts the idea of *bufferless routing* for the ring structure. The basic idea is to always route a packet to an output port regardless of whether or not that output port results in the minimal distance to the destination of the packet [22]. In our case, a *packet* means a result tuple generated by a join core, and the destination of the packet is always *N_out* port.

The routing algorithm adopted for the ring structure is based on the FLIT-BLESS (or simply BLESS) proposed in [22]. The proposed ring structure satisfies the following two constraints required for BLESS: Every *node* has 1) the same number of output ports as the number of its input ports, and 2) is reachable

from every other *nodes*.

An arbitration policy is needed to determine to which output port an incoming tuple should be forwarded. It is stated in [22] that the arbitration policy of BLESS is governed by two components: a *ranking component* and *port-selection component*. The simple oldest-first policy is adopted as a ranking policy, and for this purpose, a hop counter is added for each tuple. In every cycle, each *node* ranks all incoming tuples comparing hop counts of the tuples.

On the other hand, the port selection is based on the number of tuples in the FIFO buffers. Each buffer includes a counter that counts the number of stored tuples. In addition, these counters are connected to the ring structure. The buffer counters of *buffer k-1*, *buffer k* and *buffer k+1* are connected to the *node k*. In every cycle, the *node k* compares the counters and determines the priority of output ports according to the result of the comparison. For example, the priority of the output ports, in descending order, should be *E_out*, *W_out* and *N_out* if *counter k+1* < *counter k-1* < *counter k*.

After determining the ranks of the incoming tuples and the priority of the output ports, the *node k* considers the tuples one by one in the order of their rank (highest rank first) and assigns to the output port with highest priority that has not yet been assigned to any higher-ranked tuples. For example, let us assume there is only one incoming tuple to the *node k* at a certain time. At the same time, the priority of the output ports is, let's say, *E_out*, *W_out* and *N_out* in descending order. In this case, the incoming tuple is forwarded to *E_out* port because of its highest priority. It should be emphasized that all of the operations described above can be completed in one cycle and all of the *nodes* in the ring structure concurrently perform the same operation on each cycle in a synchronous manner.

### 6.2.3 Admission Control

The bandwidth of the output channel is not enough to transfer all result tuples when output rate is higher than the available bandwidth of the channel. In addition, some of the result tuples may be lost due to congestion (buffer overflow) losses when a large number of result tuples are generated within a short interval of time.

In order to avoid the problems with regard to the bandwidth of the output channel and the limitation of the internal buffer sizes, an admission control strategy is adopted in the proposed architecture based on Chapter 4. That is, all of the generated result tuples are transferred to the output channel by rejecting newly arrived tuples to the system when the output rate exceeds the available bandwidth of the channel, and/or an internal FIFO buffer in the merging network is close to overflow. In other words, the admission control provides a flow control mechanism between all join cores and the output port.

Each of the FIFO buffers implemented in the merging network has two state flags one of which is *full flag*. It is asserted (set to logic 1) when the corresponding buffer is almost (or completely) full. We can easily grasp the current states of the buffers by observing these flags. The admission control mechanism implemented in the handshake join operator is summarized as follows: If any of the full flags has been asserted, then

1. the newly arrived tuples are rejected,

2. and all of the join cores are suspended until all of the full flags are de-asserted (reset to logic 0).

The preceding rules guarantee that all of the result tuples generated by join cores will reach the output port of the root node (*merger1* in Figure 6.1.1) in the binary tree network.

## 6.3 Evaluation of the Proposed Implementation

The proposed architecture is implemented on a Virtex®-6 FPGA ML605 Evaluation Kit including a XC6VLX240T-1 chip as in Chapter 4. The specification of the FPGA used in the design is given in

Table 4.1. Xilinx ISE 13.1 Logic Edition is used as an FPGA development environment.

### 6.3.1 Resource Usage and Signal Delay

The hardware resource usage and clock frequency are evaluated for five different configurations. The different numbers of join cores ($2^i$ *where* $i = 1, \ldots, 5$) and corresponding merging networks are instantiated on the FPGA. The same parameters as in Chapter 4 are used during the instantiation process. The window size of each join core is set to 8 tuples. The size of each FIFO buffer in the proposed merging network is set to 8 tuples. Each input tuple consists of 64-bit data half of which is join key and the remainder is allocated for payload field. A result tuple is composed of 32-bit join key and two payload fields, a total of 96-bit data.

The maximum clock frequency of the prototype system is shown in Figure 6.3.1. The x-axis and the y-axis represent the number of join cores and the clock frequency, respectively. As shown in Figure 6.3.1, the graph is almost constant at 150MHz and the clock frequency is not declined with increased number of join cores.

The hardware resource usage is given in Table 6.1. In addition, the percentage of the overall resource consumption is shown in Figure 6.3.2. The y-axis of Figure 6.3.2 represents the percentage of the used resources. As shown in Figure 6.3.2, all of the three graphs are almost linearly increased with the increasing number of join cores. It should be also mentioned that up to 32 join cores and the corresponding merging network (with admission control) can be instantiated on the FPGA. The results of Figure 6.3.1 and Figure 6.3.2 lead us to the conclusion that the proposed design is scalable in terms of the resource usage and the signal delay.

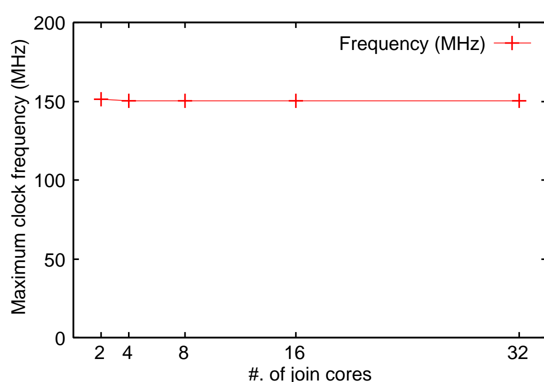



Figure 6.3.1: Maximum clock frequency.

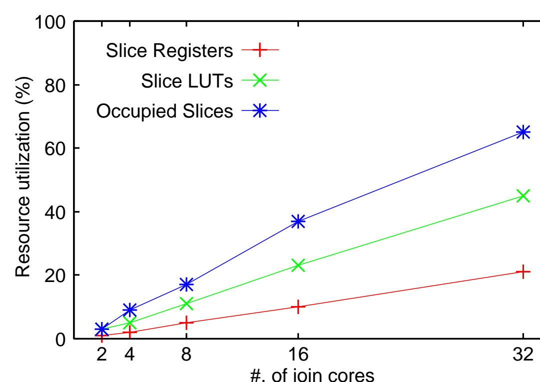



Figure 6.3.2: Overall resource consumption.

Table 6.1: Hardware resource usage

| Join cores | Slice Registers | Slice LUTs | Occupied Slices |
|---|---|---|---|
| 2 | 4,371 | 4,594 | 1,277 |
| 4 | 8,358 | 8,784 | 3,394 |
| 8 | 16,347 | 17,130 | 6,436 |
| 16 | 32,323 | 35,051 | 14,095 |
| 32 | 64,526 | 68,216 | 24,717 |

### 6.3.2 Resource Usage Comparison

Both of the baseline implementation of handshake join (presented in Chapter 4) and the proposed one with the adaptive merging network (presented in Chapter 6) utilize two types of important resource included in the FPGA. These are Block RAMs (BRAMs) and slices each of which contains *lookup tables* (LUTs) and *flip-flops* (or registers). As shown in Table 4.1, the FPGA chip (XC6VLX240T-1) includes 416 BRAMs and 37,680 slices. Each slice consists of 4 LUTs and 8 registers, resulting in total of 150,720 slice LUTs and 301,440 slice registers.

The hardware resource usage is a significant factor for evaluating the overall design implemented on the FPGA. From this point of view, two implementations are compared in terms of slice LUTs utilization, slice registers utilization, occupied slices and BRAM utilization. The different numbers of join cores from 2 to 32 are instantiated in the FPGA, and thus the two implementations are compared for five different configurations. It should be also mentioned that all of the implementations include the merging network (simple binary tree network or adaptive merging network) and the admission control mechanism.

Figure 6.3.3 indicates the results of the comparison of slice registers utilization and slice LUTs utilization. In addition, Figure 6.3.4 indicates the results of the comparison of occupied slices and BRAM utilization. In Figure 6.3.3 and Figure 6.3.4, the line labeled "baseline" represents the baseline implementation of handshake join presented in Chapter 4. Similarly, the line labeled "proposed" in Figure 6.3.3 and Figure 6.3.4 represents the proposed implementation of handshake join with the adaptive merging network. Now, let us focus on the result of each comparison separately in more detail.

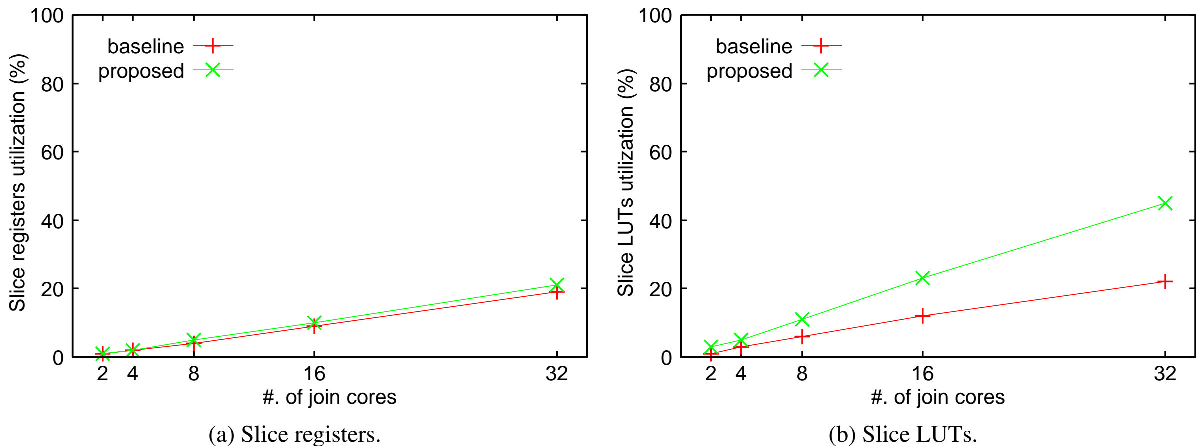



Figure 6.3.3: Comparison of slice registers and slice LUTs utilization between the baseline implementation (presented in Chapter 4) and the proposed implementation (presented in Chapter 6).

#### Slice Registers Utilization

Figure 6.3.3 (a) shows the result of the comparison of slice registers utilization between the baseline implementation (Chapter 4) and the proposed implementation (Chapter 6). The x-axis of Figure 6.3.3 (a) represents the number of join cores ($2^i$ *where* $i = 1, \ldots, 5$), and the y-axis of Figure 6.3.3 (a) represents slice registers utilization in percentage.

As shown in Figure 6.3.3 (a), both lines labeled "baseline" and "proposed" linearly increase with almost the same slope. The result is reasonable because slice registers are mainly used for implementing join cores (especially windows implemented over the join cores). As mentioned in Chapter 4, each segment of the windows of the input streams $R$ and $S$ is implemented as large *shift registers* that hold

tuple data of the each stream. When it comes to implementing the shift registers in the FPGA, the slice registers are utilized. In addition, both implementations of handshake join presented in Chapter 4 and Chapter 6 adopt the same approach for implementing each segment of the windows included in the join cores. This explains why both of the lines increase with almost the same slope.

#### Slice LUTs Utilization

Figure 6.3.3 (b) shows the result of the comparison of slice LUTs utilization between the baseline implementation (Chapter 4) and the proposed implementation (Chapter 6). The x-axis of Figure 6.3.3 (b) represents the number of join cores ($2^i$ *where* $i = 1, \ldots, 5$), and the y-axis of Figure 6.3.3 (b) represents slice LUTs utilization in percentage.

As shown in Figure 6.3.3 (b), both lines labeled "baseline" and "proposed" almost linearly increase with the increasing number of join cores. Contrary to Figure 6.3.3 (a), however, the two lines increase with different slopes. This leads to an important difference in slice LUTs utilization between the two implementations, especially for a large number of join cores. For example, the proposed implementation of the handshake join with the adaptive merging network requires 23 percent more slice LUTs compared to the baseline implementation of the handshake join when the number of join cores equals to 32.

This difference comes from the complexity of the design of the adaptive merging network (especially the routing algorithm implemented in the ring structure which require many comparators). As explained before, each *node* of the ring structure included in the adaptive merging network has a total of six ports as shown in Figure 6.2.1. The routing algorithm adopted for the ring structure is based on the idea of *bufferless routing*, requiring two components: a *ranking component* and *port-selection component*.

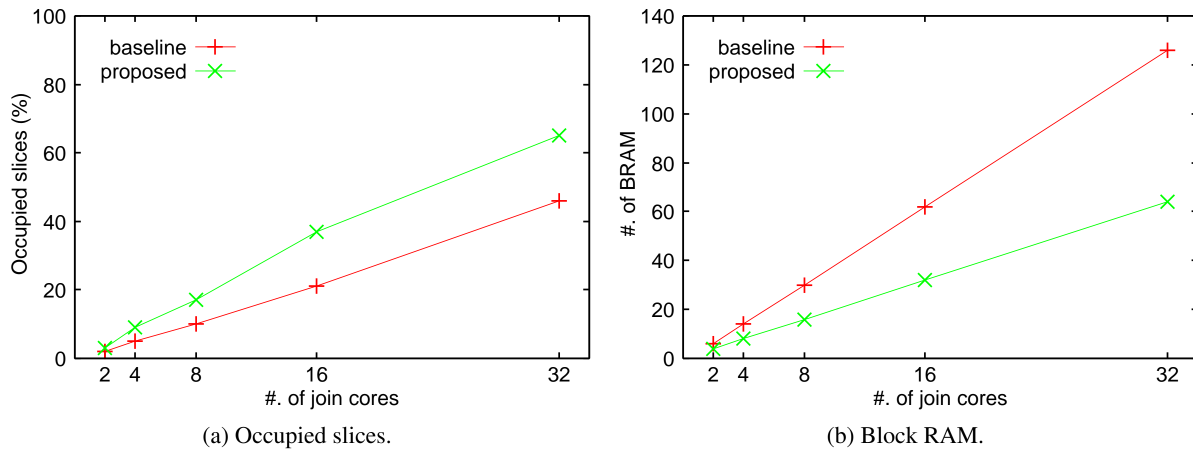
In every cycle, the *ranking component* of each *node* ranks all incoming tuples by comparing hop counts of the tuples. At the same time that the incoming tuples are compared to each other, the *port-selection component* of each *node* compares three buffer counters connected to the *node*, determining the priority of output ports according to the result of the comparison. All of the operations are completed in just one cycle and all *nodes* in the ring structure concurrently perform the same operation on each cycle in a synchronous manner, by taking advantage of the parallelism that FPGA hardware provides.

When it comes to implementing the comparators in the FPGA, the slice LUTs are utilized. In general, slice LUTs are used to implement arbitrary Boolean-valued functions including comparison that is the case for the ring structure included in the adaptive merging network. As shown in Figure 4.3.2, the merging network of the baseline implementation of the handshake join is composed of the mergers and their corresponding connections whereas the proposed implementation of the handshake join with the adaptive merging network (Figure 6.1.1) includes the ring structure requiring the additional LUTs to implement the functionality of the routing logics (*i.e.*, *ranking component* and *port-selection component*). This explains why the proposed implementation requires more slice LUTs than the baseline implementation. We can regard the difference of the slice LUTs utilization as necessary overhead in order to implement the adaptive merging network.

#### Occupied Slices

Figure 6.3.4 (a) shows the result of the comparison of occupied slices between the baseline implementation (Chapter 4) and the proposed implementation (Chapter 6). The x-axis of Figure 6.3.4 (a) represents the number of join cores ($2^i$ *where* $i = 1, \ldots, 5$), and the y-axis of Figure 6.3.4 (a) represents occupied slices in percentage.

As shown in Figure 6.3.4 (a), both lines labeled "baseline" and "proposed" almost linearly increase with the increasing number of join cores. Contrary to Figure 6.3.3 (a), the two lines in Figure 6.3.4 (a) increase with different slopes similarly to Figure 6.3.3 (b). This leads to an important difference in




(a) Occupied slices.

(b) Block RAM.

Figure 6.3.4: Comparison of occupied slices and BRAM utilization between the baseline implementation (presented in Chapter 4) and the proposed implementation (presented in Chapter 6).

the number of occupied slices between the two implementations, especially for a large number of join cores as in Figure 6.3.3 (b). For example, the proposed implementation of the handshake join with the adaptive merging network utilizes 19 percent more slices compared to the baseline implementation of the handshake join when the number of join cores equals to 32.

Slices are the fundamental components included in an FPGA, used as the basic building-block components to implement desired functionality on the FPGA. As mentioned before, each slice includes LUTs and registers. In general, slice LUTs are required to implement Boolean-valued functions and slice registers act as temporary storage areas inside the FPGA. The FPGA used in present work has 37,680 slices each of which includes 4 slice LUTs and 8 slice registers.

When it comes to synthesizing the logic designs onto the FPGA using Xilinx ISE, all of the LUTs and registers required to implement the design are located inside the slices. In other words, all LUTs and registers are packed into slices as slice LUTs and slice registers, respectively. During the packing process, all of the LUTs and registers are assigned to the slices, but not necessarily utilizing all slice LUTs and slice registers in a slice.

For example, a slice with 4 slice LUTs and 8 slice registers can be used for just one register. In this case, 4 LUTs and 7 register of the single slice are not utilized; however, any slice that is used even partially (in this example, only one register) is counted in the *occupied slices*. That's why the percentage of the occupied slices (Figure 6.3.4 (a)) is greater than the slice registers utilization (Figure 6.3.3 (a)) and/or the slice LUTs utilization (Figure 6.3.3 (b)). For example, the proposed implementation of the handshake join with the adaptive merging network occupied 65 percent of total slices even though it utilizes only 21 percent of the slice registers and 45 percent of the slice LUTs when the number of join cores equals to 32. Similarly, the baseline implementation of the handshake join occupied 46 percent of total slices even though it utilizes only 19 percent of the slice registers and 22 percent of the slice LUTs when the number of join cores equals to 32.

As explained before, the proposed implementation requires almost the same amount of slice registers as the baseline implementation. On the other hand, however, it needs more slice LUTs than the baseline implementation, particularly 23 percent more slice LUTs compared to the baseline implementation of the handshake join when the number of join cores equals to 32. As a result, the proposed implementation uses 19 percent more slices compared to the baseline implementation mainly because of the significant difference in the slice LUTs utilization between the proposed implementation and the baseline implemen-

tation. This explains why the proposed implementation requires more slices (or results in more *occupied slices*) than the baseline implementation.

#### Block RAM Utilization

Figure 6.3.4 (b) shows the result of the comparison of Block RAM (BRAM) utilization between the baseline implementation (Chapter 4) and the proposed implementation (Chapter 6). The x-axis of Figure 6.3.4 (b) represents the number of join cores ($2^i$ *where* $i = 1, \ldots, 5$), and the y-axis of Figure 6.3.4 (b) represents the total number of BRAMs included in each of the handshake join operator.

As shown in Figure 6.3.4 (b), both lines labeled "baseline" and "proposed" linearly increase with the increasing number of join cores. The two lines in Figure 6.3.4 (b) increase with different slopes, and this leads to an important difference in the number of BRAMs between the two implementations, especially for a large number of join cores. It should be emphasized that the proposed implementation of the handshake join with the adaptive merging network requires fewer BRAMs than the baseline implementation of the handshake join. It can be easily noticed that the result presented in Figure 6.3.4 (b) is contrary to what is shown in Figure 6.3.3 (b) (slice LUTs utilization) or Figure 6.3.4 (a) (occupied slices). For example, the proposed implementation requires only 64 BRAMs whereas the baseline implementation needs 126 BRAMs when the number of join cores equals to 32. In this case, the baseline implementation consumes nearly two times as many BRAM resources as the proposed implementation consumes.

As explained before, the baseline implementation of the handshake join suffers from the structural disadvantage concerning efficient use of the FIFO buffers implemented based on BRAMs. As shown in Figure 4.3.2, each merger includes a FIFO buffer and there is only one path from each join core to the output port. This is because join cores are located at leaf nodes of the binary tree network and all of the result tuples are forwarded towards the root node (*i.e.*, *merger1*) of the tree. The baseline implementation adopts the binary tree network (Figure 4.3.2) as its merging network, and this approach leads to a bottleneck for overall system when the output rate of a join core significantly differs from those of others.

In order to alleviate the problem, the adaptive merging network is proposed as indicated in Figure 6.1.1. As shown in Figure 6.1.1, the FIFO buffers are omitted from both join cores and mergers. Instead, a new layer of *nodes* and the FIFO buffers are located between join cores and mergers. This approach significantly reduces the total number of BRAMs required to implement the FIFO buffers. That's why the proposed implementation of the handshake join with the adaptive merging network consumes fewer BRAM resources than the baseline implementation.

### 6.3.3 Performance Evaluation

In the performance evaluation, the same evaluation model as in Chapter 4 is adopted as shown in Figure 4.5.3. Before starting the join operation, a number of input tuples are generated according to different match rates which indicate the ratio of the tuples satisfying the join condition. After that, input tuples are transferred to the handshake join operator and join operation is applied to the input tuples in a continuous manner.

The join operator generates result tuples if the join condition is satisfied while processing the input tuples. The result tuples are transferred to the output port as a single stream by the merging network and they are stored to the output buffer. It should be noted that all of the result tuples generated by join cores are transferred to the output buffer owing to the admission control mechanism. This is confirmed by counting the number of results stored in the output buffer.

Figure 6.3.5 shows the number of result tuples, and the total number of cycles required to complete the join operation. In this evaluation, the handshake join operator consists of 16 join cores, and the size

of the input buffer is set to 128 tuples per input stream. The input tuples generated according to the match rate are located in the input buffer in random order.

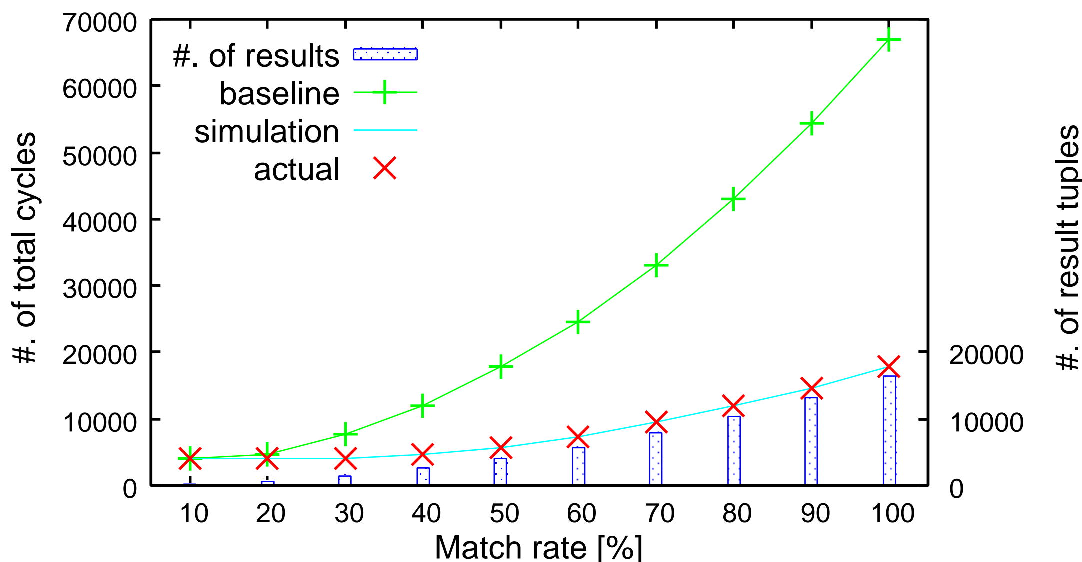



Figure 6.3.5: The number of result tuples, and the total number of cycles required to complete the operation.

The x-axis represents the match rate from 10% to 100%. The y-axes in left and right represent the number of total cycles and the number of result tuples, respectively. The line labeled “baseline” is the performance estimation of the handshake join presented in Chapter 4. The same number of result tuples is generated by both of the join operators, and it is indicates as a bar chart in Figure 6.3.5.

The proposed model is evaluated by both a cycle-accurate simulator and the FPGA platform that is used to implement the architecture. Precisely the same results, which are labeled “simulation” and “actual”, are obtained as shown in Figure 6.3.5. Results indicate that the baseline increases sharply if the match rate is increased. By contrast, the total number of cycles required for the proposed architecture only increases in accordance with the number of result tuples.

The input throughput performance is shown in Figure 6.3.6. This is the maximum throughput data rate that can be handled by each join operator without dropping any tuples. In this evaluation, the handshake join operator consists of 64 join cores, and the size of the input buffer is set to 512 tuples per input stream. The lines labeled “baseline” and “nested loop join” represent the handshake join (Chapter 4) and the nested loops-style join [5], respectively. It should be noted that the size of the input buffer for the nested loops-style join [5] is also set to 512 tuples for regulating the condition.

The proposed model is evaluated by a cycle-accurate simulator with input streams of two different characteristics. The input tuples generated according to the match rate are located in the input buffer as follows:

1. in random order,

2. and as burst input (consecutive tuples that satisfy the join condition).

The results are labeled “proposed model (random input)” and “proposed model (burst input)” in Figure 6.3.6.

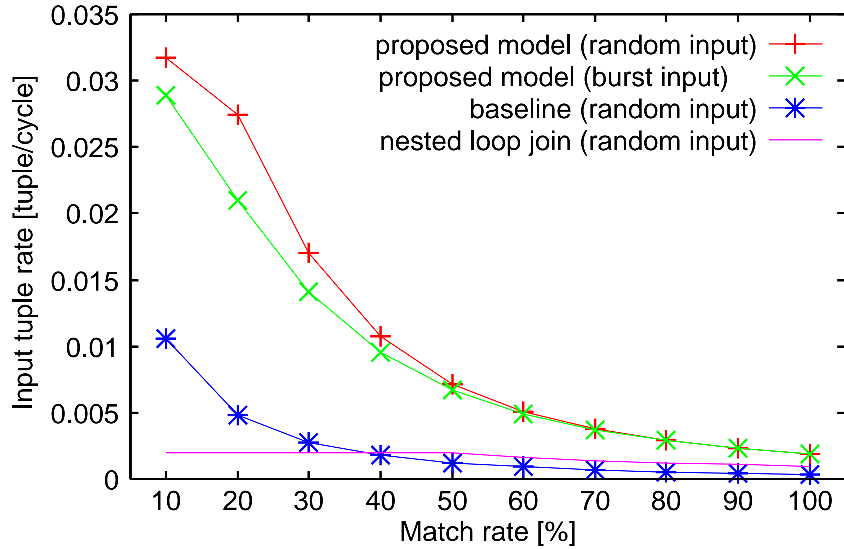



Figure 6.3.6: Input throughput with 64 join cores.

As shown in the graph, the baseline implementation (Chapter 4) can achieve higher throughput rate than nested loops-style join [5], only when the match rate is lower than 40%. On the other hand, the proposed model can achieve far higher throughput rate than nested loops-style join [5] even if the match rate is increased. In particular, the proposed implementation achieves more than 5.2 times higher throughput compared to the baseline implementation (Chapter 4) at the highest match rate (*i.e.*, 100%). It also outperforms the nested loops-style join [5], demonstrating up to 16.3 times higher throughput without dropping any tuples. Furthermore, the proposed architecture can handle high input rates compared to the implementations of Chapter 4 and [5] despite burst inputs which can be considered as the worst case. To the best of our knowledge, this is the best performance for handshake join operator implemented on an FPGA. These data lead us to the conclusion that the proposed architecture can considerably outperform both the handshake join (Chapter 4) and the nested loops-style join [5].

# Chapter 7

# Conclusion

Chapter 7 gives conclusions and identifies future work.

## 7.1 Summary

In this thesis, a complete design and implementation of handshake join is presented based on [1]. In handshake join, it is necessary to take into account the result merging logic, and the problems with regard to the limitation of the bandwidth of the output channel and the size of the buffers included in join cores and mergers. The three design issues mentioned in the introduction are addressed by the proposed design including the binary tree network and the admission control mechanism. The proposed additional mechanism contributes to solving the buffer overflow problem in the handshake join operator.

The proposed implementation is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the throughput performance. The result shows that the proposed implementation achieves scalability up to 64 cores as mentioned in [1], even though it includes the merging network and the admission control mechanism. The performance evaluation results show that the handshake join handles considerably high input rate compared with nested loops-style join [5] when the match rate is low. Moreover, simulation results indicate a new intuition regarding static and adaptive tuning of the FIFO buffers included in join cores and mergers.

Furthermore, this research proposes an adaptive merging network for hardware implementation of the handshake join by examining the weakness of the naively implemented merging network based on [1]. Moreover, a complete handshake join operator is implemented with the proposed merging network on an FPGA. Result collection is a crucial issue for the handshake join operator especially at high output rates since the merging network becomes an overwhelming bottleneck for overall performance. In fact, it is an important limiting factor for the design of handshake join hardware. The suitable network architecture and the careful design are key requirements to improve the operation performance; therefore, the significantly improved network structure is proposed in order to achieve far higher throughput performance.

The improved architecture is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the throughput performance. The result of evaluation for clock frequency shows that the proposed architecture achieves scalability up to 32 cores as mentioned in [1], even though it includes the adaptive merging network with the admission control mechanism. The performance evaluation results show that the proposed architecture handles considerably high input rate compared with the baseline implementation of handshake join (presented in Chapter 4) and the nested loops-style join [5].

## 7.2 Future Work

Future work is as follows. First of all, the current design of join cores will be improved in several aspects. In the proposed design, each segment of the windows for input streams is implemented using shift registers. As a result, the total size of the window is severely limited by the available hardware resources. An alternative implementation technique should be considered to handle large windows. Furthermore, a load balancing strategy for join cores can be implemented to enhance the overall performance. For example, if certain cores are overloaded, the overloaded cores would transfer some of their loads to the neighboring cores.

Secondly, the proposed implementation of handshake join tolerates output latency in order to handle higher input rates. The latency, however, is not mainly related to the execution strategy (whether or not join processes are executed in parallel). The longer latency occurs in the merging network after results are produced in each join core; therefore, an improved network structure can offer much better latency characteristics than the proposed one.

Finally, the performance of the proposed implementation will be compared to another implementation of window joins (*e.g.*, CellJoin). It should also be evaluated through practical application, determining suitable size of the FIFO buffers included in the handshake join hardware.

# Acknowledgement

I would like to express my sincere gratitude to my advisor Prof. Tsutomu Yoshinaga for his supervision throughout my study and research as well as for his patience, motivation and enthusiasm. His guidance helped me in all the time of research and writing of this thesis.

Besides my advisor, I would like to thank the other members of my thesis committee: Prof. Hiroyoshi Morita and Dr. Hidetsugu Irie, whose helpful suggestions increased readability and reduced ambiguity.

My sincere thanks also goes to Dr. Takefumi Miyoshi and Dr. Hideyuki Kawashima for their encouragement, insightful comments, and helpful advices.

Last but not the least, I would like to thank my family, especially my parents for giving birth to me at the first place and supporting me throughout my life.

# References


[1] Jens Teubner and René Müller. How soccer players would do stream joins. In *SIGMOD Conference*, pp. 625–636, 2011.

[2] Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pp. 341–352, 2003.

[3] Bugra Gedik, Rajesh Bordawekar, and Philip S. Yu. Celljoin: a parallel stream join operator for the cell processor. *VLDB J.*, Vol. 18, No. 2, pp. 501–519, 2009.

[4] Theodore Johnson, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. Streams, security and scalability. In *DBSec*, pp. 1–15, 2005.

[5] Yuta Terada, Takefumi Miyoshi, Hideyuki Kawashima, and Tsutomu Yoshinaga. A consideration of window join operator over data streams by using FPGA (in Japanese). In *IEICE Tech. Rep.*, pp. 181–186, 2011.

[6] René Müller, Jens Teubner, and Gustavo Alonso. Data processing on FPGAs. *PVLDB*, Vol. 2, No. 1, pp. 910–921, 2009.

[7] Mohammad Sadoghi, Hans-Arno Jacobsen, Martin Labrecque, Warren Shum, and Harsh Singh. Efficient event processing through reconfigurable hardware for algorithmic trading. *PVLDB*, Vol. 3, No. 2, pp. 1525–1528, 2010.

[8] Mohammad Sadoghi, Harsh Singh, and Hans-Arno Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *DaMoN*, pp. 27–32, 2011.

[9] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H.A. Jacobsen. Multi-query stream processing on FPGAs. In *ICDE*, 2012.

[10] Jens Teubner, René Müller, and Gustavo Alonso. Frequent item computation on a chip. *IEEE Trans. Knowl. Data Eng.*, Vol. 23, No. 8, pp. 1169–1181, 2011.

[11] René Müller, Jens Teubner, and Gustavo Alonso. Streams on wires - a query compiler for FPGAs. *PVLDB*, Vol. 2, No. 1, pp. 229–240, 2009.

[12] Takefumi Miyoshi, Hideyuki Kawashima, Yuta Terada, and Tsutomu Yoshinaga. A coarse grain reconfigurable processor architecture for stream processing engine. In *FPL*, pp. 490–495, 2011.

[13] Nesime Tatbul, Ugur Çetintemel, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *VLDB*, pp. 309–320, 2003.

[14] Jiang bo Qian, Hong bing Xu, Yisheng Dong, Xue jun Liu, and Yong li Wang. FPGA acceleration window joins over multiple data streams. *Journal of Circuits, Systems, and Computers*, Vol. 14, No. 4, pp. 813–830, 2005.

[15] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proc.*, pp. 256–282. SIAM, 1979.

[16] H. T. Kung and Philip L. Lehman. Systolic (VLSI) arrays for relational database operations. In *SIGMOD Conference*, pp. 105–116, 1980.

[17] Sailesh Krishnamurthy, Michael J. Franklin, Jeffrey Davis, Daniel Farina, Pasha Golovko, Alan Li, and Neil Thombre. Continuous analytics over discontinuous streams. In *SIGMOD Conference*, pp. 1081–1092, 2010.

[18] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. Semantics and evaluation techniques for window aggregates in data streams. In *SIGMOD Conference*, pp. 311–322, 2005.

[19] CERT. Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks, 1996.

[20] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the Borealis stream processing engine. In *CIDR*, pp. 277–289, 2005.

[21] Yuval Tamir and Gregory L. Frazier. Dynamically-allocated multi-queue buffers for VLSI communication switches. *IEEE Trans. Computers*, Vol. 41, No. 6, pp. 725–737, 1992.

[22] Thomas Moscibroda and Onur Mutlu. A case for bufferless routing in on-chip networks. In *ISCA*, pp. 196–207, 2009.

# List of Publications Related to the Thesis

- **Yasin Oge**, Takefumi Miyoshi, Hideyuki Kawashima, and Tsutomu Yoshinaga.
  "An Implementation of Handshake Join on FPGA",
  International Conference on Networking and Computing (ICNC'11), pp. 95–104, 2011.

- **Yasin Oge**, Takefumi Miyoshi, Hideyuki Kawashima, and Tsutomu Yoshinaga.
  "Design and Implementation of a Handshake Join Architecture on FPGA",
  IEICE Trans. ED, Special Section on Parallel and Distributed Computing and Networking [accepted for publication].

- **Yasin Oge**, Takefumi Miyoshi, Hideyuki Kawashima, and Tsutomu Yoshinaga.
  "Design and Implementation of a Merging Network Architecture for Handshake Join Operator on FPGA",
  IEEE 6th International Symposium on Embedded Multicore SoCs [accepted for publication].