An Implementation of Handshake Join on FPGA

Yasin Oge*, Takefumi Miyoshi*, Hideyuki Kawashima†, and Tsutomu Yoshinaga*
* Graduate School of Information Systems, University of Electro-Communications, Japan
1-5-1 Chofugaoka, Chofu-shi, Tokyo 182-8585, Japan
E-mail: oge@comp.is.uec.ac.jp, {miyoshi, yosinaga}@is.uec.ac.jp
† University of Tsukuba, Japan
1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan
E-mail: kawasima@cs.tukuba.ac.jp

***Abstract*—This paper shows an implementation of handshake join on field-programmable gate array (FPGA). Handshake join is one of stream join algorithms, proposed by Teubner and Mueller. It can support very high degrees of parallelism and attain unprecedented success in throughput speed in order to achieve efficient support for window-based join in streaming databases. In handshake join, it is necessary to take into account the problems with regard to the capacity of the output channel and the limitation of the internal buffer sizes, in order to apply join operation to input tuples efficiently in a correct manner. However, the implementation has not necessarily clarified in detail yet in their paper. In this paper, to solve the issues, we propose the merging network and the admission controller. Then we evaluate the architecture in terms of the hardware resource usage, the maximum clock frequency, and the operation performance.**

I. Introduction

Nowadays, stream data processing systems demand more functionality than in previous years. Many of these data processing tasks, such as financial analysis, traffic monitoring and data processing in sensor networks, are required to handle the huge volume of data with certain time restrictions for each specific application. Consequently, low-latency and high-throughput processing are key requirements of such kind of systems that process unbounded, continuous input streams rather than fixed-size stored data sets.

Most of today's modern general-purpose relational database management systems (DBMSs) have been added powerful, complicated features; however, all of these DBMSs should provide basic set oriented operations or traditional set operations including union, intersection, difference and Cartesian product. In addition to these basic operations, relational databases support special relational operators such as join, selection, projection and division. Likewise, streaming databases also support similar operation sets and one of these fundamental operations is called stream join or window join [1] that introduces window semantics besides value-based join predicates.

Streaming databases deal with unbounded, infinite streams of data that have to be processed immediately for real time applications. It is a well-known fact that infinite inputs cause semantic problems when join operator is applied to unbounded input streams of data due to the nature of the join operation. This is because a join operator has to take into account the all combination of the input tuples.

In order to solve the problem mentioned above, the window semantic is introduced for practical applications in streaming databases. That is to say, a finite subset of the unbounded input data is defined as "a window", and join operation is evaluated over windows which are finite subsets of the streams.

So far, the dominant strategy for executing window-based stream join operation is mostly sequential. However, stream joins are fundamental and costly operations in streaming databases. Therefore, maximizing the parallelism of the stream join operation is crucial in order to increase the performance of stream joins.

Teubner and Mueller have provided new insight into stream join algorithm, and proposed a novel approach, named handshake join, which is a stream join implementation that can support very high degrees of parallelism and attain unprecedented success in throughput speed in order to achieve efficient support for window-based join in streaming databases[2]. It is stated that handshake join naturally leverages available hardware parallelism, which they demonstrate with an implementation on a modern multi-core system and on top of FPGA. Implementation of handshake join on top of a modern multi-core CPU, in software side, considerably outperforms CellJoin[3] which is another well-known implementation of the window-based join for the Cell processor. On the other hand, in hardware side, it is mentioned that the hardware implementation is scalable, because the maximum clock frequency is almost unaffected regardless of increased number of cores. However, the hardware implementation has not necessarily clarified in detail yet in [2]. Therefore, it is not clarified whether handshake join is efficient to implement join operator on FPGA or not.

It is a fact that the output generation rate, which is a measurement of how frequent the output tuples are generated by join cores, may exceed the input rate due to the nature of the parallelized execution of the handshake join operator. In handshake join, it is necessary to take into account following three points in order to apply join operation to input tuples efficiently in a correct manner:

1) a mechanism to transfer all generated result tuples without any loss of results,
2) a mechanism to interrupt data injection from input buffer when the join operator is unable to keep up with the rates of the input streams,

3) and a scalable mechanism that collects result tuples generated by all join cores as a single stream.

Since these mechanisms require large buffers and combinational logics, the hardware resource usage and the signal delays are not negligible for the overall implementation. Obviously, these are difficult issues although they are ignored in the original paper of handshake join[2].

In order to clarify the issues pointed out above, in this paper, we discuss details of handshake join architecture onto FPGA, and then evaluate the hardware resource usage, the maximum operating frequency, and the throughput performance. It should be noted that the proposed architecture of handshake join operator is implemented on a FPGA and evaluated as a case study. In our view, the major contribution of the paper is that the study clarifies the details for the implementation of handshake join operator and the proposed architecture is verified by implementing on FPGA board. As a result, the paper presents the performance comparison with the other implementation in addition to the speed and the logic size of the implemented handshake join hardware. To the best of our knowledge, this is the first paper that implements handshake join operator on FPGA.

The rest of the paper is organized as follows: Section 2 gives a background and briefly reviews the previous work. Section 3 introduces handshake join and the implementation issues onto FPGA. Section 4 proposes the details of handshake join architecture to implement it onto FPGA. Then, Section 5 evaluates our proposed architecture. Finally, Section 6 gives our conclusions and identifies future work.

II. Background and Related Work

Due to increasing demand for processing unbounded, continuous input data streams, DBMS researchers have expanded the data processing paradigm from the traditional store and then process model towards the stream-oriented processing model. Accordingly, an extensive range of research is taken place for addressing and resolving the new problems that come about because of the nature of stream-oriented processing model.

Because of the tight response-time restriction for each specific application, how to implement the stream join operator is one of the most challenging tasks in streaming databases. In addition, stream join operation needs a heavy computational cost. Therefore, efficient implementation method of the stream join operator is required for stream databases in order to overcome the computational cost and meet the time requirement. Consequently, acceleration of the stream join operation is one of the most significant research issues regarding stream joins.

It is a well-known fact that the modern CPU architectures are subjected to crucial restriction and limitations, some of which are the high latency occurred when getting data in and out of the system, and memory wall that can cause an overwhelming bottleneck in the entire system. Nevertheless, as stated in the previous section, low-latency and high-throughput data processing are key requirements of streaming database systems. Thus, in order to figure out this problem, FPGAs, which are not bound by the von Neumann architecture, can be proposed as an alternative platform to implement the stream join operation. In fact, FPGAs are considered as a possible solution for the inherent limitations of classical CPU-based system architectures[4][5][6].

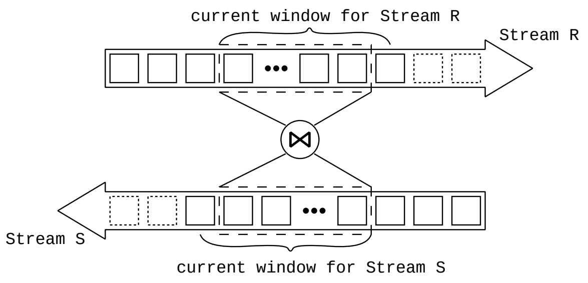


Fig. 1. The basic idea of the handshake join. Tuples from the two input streams are flowed in opposite direction.

Terada et al.[7] suggest an implementation technique of window join operator on the FPGA platform in order to improve the performance and achieve high-throughput with low-latency. Nonetheless, concurrent execution of only two matching processes, which compare key values and perform join operation have been accomplished although they have tried to extract parallelism from window join operation. This is because the widely used approach, including Terada et al. [7], to perform the computation of the window-based join is nearly sequential as defined in the operation semantics.

In this work, we study an implementation of parallel stream join, namely handshake join, which is extremely tractable for parallelized calculation on the FPGA platforms, and evaluate the performance of the handshake join by using the dedicated hardware implemented on a FPGA board.

III. Handshake Join Implementation

A. Handshake Join

The basic idea of the handshake join, which is a brand new method for executing window based join introduced by Teubner and Mueller[2], is to consider two input streams which are allowed to flow in opposite direction as shown in Fig. 1. From this point of view, we obtain significant advantages regarding parallelization and scalability. Consequently, we can easily extract parallelism from window join semantics and this enables us to implement parallelized execution of stream join.

The previous approaches of implementing stream join, including the implementation technique suggested by Terada et al.[7], are generally based on the three-step procedure, proposed by Kang et al.[1], which results in the sequential execution of the nested loop join in the most common case.

In reference [1], the three-step procedure, which introduces windowing semantics and includes implicit semantics for window-based stream joins, is described as follows: a window join operator takes two streams of tuples as inputs; let us say stream R and stream S. The output is also a stream of tuples that consist of tuples (r, s), where r is from stream R, s is from stream S, such that (i) r and s satisfy the join predicate, and (ii) r was in the active window for stream R at the same time that s was in the active window for S.

One of the crucial and fundamental problems of the three-step procedure is that the approach adopted by Kang et al. is not suitable for parallel execution of the stream join on the modern system architectures. In fact, although the locality of the data at the each processing core is highly significant for achieving optimal performances in many-core systems, the nature of the window join operation is contrary to the local availability of the data since all tuples in one of the windows have to be compared with the all tuples in the opposite side window.

The main problem is that the traditional three-step procedure gives rise to control flow problem that brings about scalability issues when the number of processing cores increases. In order to resolve this problem, the distributed data flow-style processing model without a dedicated centralized coordinator, which manages computing resources as well as the data to be processed, is proposed with the handshake join approach.

In Fig. 1, each rectangular box represents a tuple from two input streams that are named stream R and stream S. If we presuppose each window is tuple-based window, all tuples in the window shift one-step to the side whenever a new tuple arrives at the entrance of the respective windows, so that the oldest tuple always flows out of the each window. In addition, both of the stream windows are oriented side by side in order that tuples from the two input streams flow in opposite direction. When two streams of tuples r and s, where r is from stream R and s is from stream S, encounter each other, the join condition is evaluated over r and s. As a result, if the join condition is satisfied, a result tuple (r, s) is generated as a tuple of the output stream. With this approach, a great number of processing elements or cores are able to evaluate the join condition simultaneously (at the same time), allowing us to parallelize the stream join operation over available computing resources without a central coordinator that manages data and available resources.

It is indicated in Teubner et al.[2] that despite the fact that an altogether new approach is adopted in handshake join, it still produces exactly the same output tuples as classical window-based stream join procedure. As a result, the handshake join can be deliberated as safe substitute for current window join implementations.

The parallelization of the handshake join calculation is illustrated in Fig. 2. In these illustrations, how handshake join can be executed in a parallel manner with two and three processing units (or cores) is shown respectively (see Fig. 2 (a) and (b)). As shown in Fig. 2, by increasing the number of processing units, the degree of parallelism can be easily increased a higher level than ever achieved before. Since each of the processing units 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. Hence, theoretically, it can be readily scaled

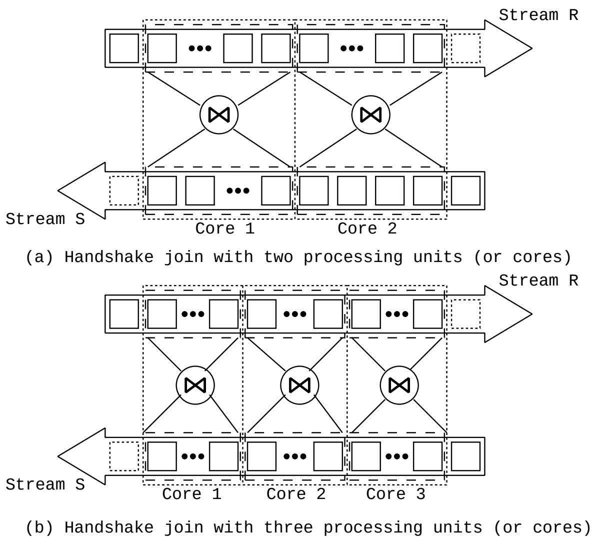

Stream R
Stream S
Core 1
Core 2
(a) Handshake join with two processing units (or cores)
Stream R
Stream S
Core 1
Core 2
Core 3
(b) Handshake join with three processing units (or cores)

Fig. 2. The parallelization of the handshake join calculation. Each of the cores evaluates its own segment of the both windows.

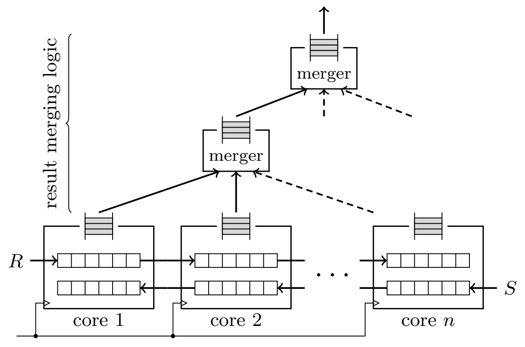


Fig. 3. Overview of the handshake join with tuple-based windows for FPGA implementation (figure adopted from [2]).

up in order to support large window sizes, achieve high throughput rates, and/or handle compute intensive functions of the join conditions.

B. Implementation Issues of Handshake Join

Fig. 3 (adopted from [2]) illustrates the general overview of the handshake join with tuple-based window for FPGA implementation. As shown in Fig. 3, join cores, which evaluate the join condition over tuples r and s in their respective windows and perform window join calculation simultaneously, are aligned side by side so that the tuples of the stream R and S flow in opposite direction. Furthermore, it can be easily noticed that the windows of the two input streams, R and S, divided into n sub-windows respectively over n join cores.

Based on the Fig. 3, there might be several issues that should be considered and resolved so as to implement the handshake join hardware on FPGAs. However, three significant and fundamental problems have to be dealt with in order to realize the handshake join hardware.

First of all, result collection is one of the most important issues when realizing handshake join on the FPGA platforms. As illustrated in Fig. 3, result merging logic is placed on top of the join cores. Nonetheless, the architecture of the result merging logic is not clear enough thought it is stated in [2] that a merging network merges all sub-results that are generated by each join core into the final join output.

Secondly, another significant issue is the capacity of the output channel. It is a fact that the output generation rate, which is a measurement of how frequent the output tuples are generated by join cores, may exceed the input rate due to the nature of the parallelized execution of the handshake join operation. Moreover, the output rate may even surpass the capacity of the output channel and in this case, the capacity of the output channel is not enough to transfer all generated result tuples.

Thirdly and finally, the limitation of the internal buffer sizes is considered as another critical problem. Provided that the number of result output tuples is more than the amount of input tuples, some of the output tuples overflow out of the internal buffers and thus, correct results are permanently lost especially when a large number of output tuples are produced in each join core at the same time (or within a short interval of time).

In this work, we will present the possible solutions of the above issues and show how our proposed architecture overcomes above-mentioned problems.

C. *Implementation Strategy of Handshake Join*

As previously mentioned, three issues which are result collection, capacity of the output channel, and limitation of the internal buffer sizes should be resolved. Unless these issues are handled properly, the implementation of the handshake join operator could not be completed.

Because it is necessary to take into account the above issues, the following components and the mechanism will be introduced in our proposed architecture in order to complete the implementation of the handshake join operator:

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

Join core and merger components are shown in Fig. 3. These are fundamental components for applying join operation to tuples of the input streams and generating a single output stream.

Merging network is a result merging logic that determines how merger circuits and join cores should be connected each other. The merging network should be scalable so that it is able to collect result tuples generated by join cores when the number of join cores is increased.

Admission control mechanism enables us to transfer all generated result tuples to output port without any loss of results. Moreover, this mechanism interrupts tuples of the input streams when handshake join operator is unable to handle the new input tuples owing to the relatively high input rate of the input streams.

We have given an overview of the general concept of the proposed architecture so far. In the following section, we will describe the details of the each component and the mechanism.

IV. Architecture of Handshake Join

The architecture of handshake join that will be presented is based on Fig. 3. As stated in previous sub-section, 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 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 Fig. 3, in our proposed architecture, 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 Fig. 3, a hardware implementation of join cores and merger circuits is need to be provided in order to complete the implementation of the handshake join operation. 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 the Fig. 3. 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 Fig. 3, and given the general ideas of the handshake join. Now, let us describe how these circuits are implemented in our proposed architecture in further detail.

A. *Join core*

The most fundamental circuit in our architecture is, of course, join processor or 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 are one-bit "valid flag" fields 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 data in respective tuple field, whereas if a valid flag is set 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 Fig. 4. 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 Fig. 4 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 set 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.

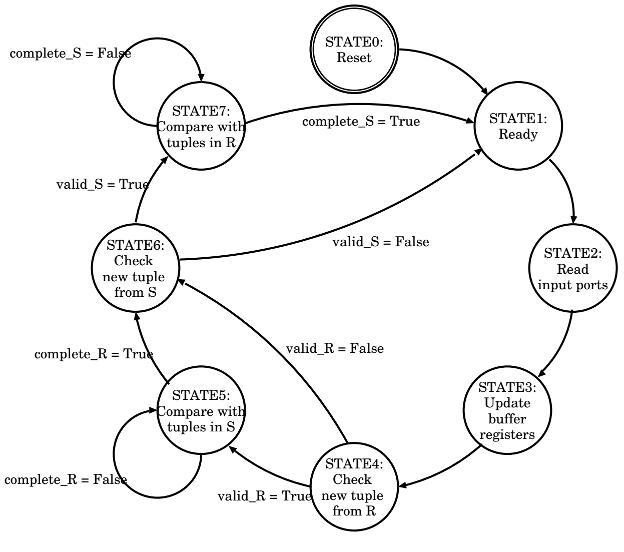
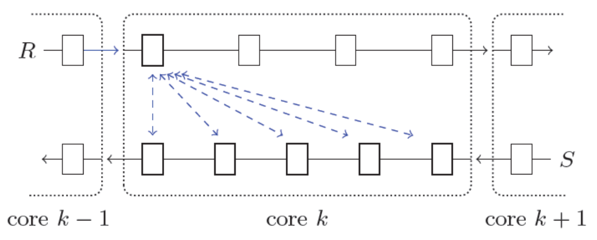


Fig. 4. State transition diagram of the join core circuits.

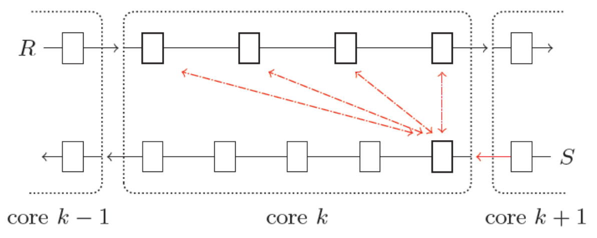
Consequently, input buffer registers and corresponding valid flags have been updated accurately and these buffers are ready to be processed at the beginning of the next state i.e. **STATE4**.

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 the Fig. 4. The next state will be determined as follows 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 set 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 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 [2]). 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



(a) Tuple from stream R entered segment.



(b) Tuple from stream S entered segment.

Fig. 5. 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 [2]).

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 [2]. 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 architecture, we have used immediate scan strategy with a nested loop join implementation as in [7].

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 Fig. 5 (adopted from [[2]]). 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 the join core k, the immediate comparison will be triggered respectively on the corresponding segments of the windows in the join core k (see Fig. 5). Accordingly, as shown in the illustration given in Fig. 5 (a), after entering the segment k of the window for stream R, a newly entered tuple is compared at once with the all tuples of stream S that are already in the same segment of the window. Fig. 5 (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 the all tuples of stream R that are already in the same segment of the window as shown in Fig. 5(b).

Let us come back to the **STATE5** in Fig. 4. As mentioned before, we have adopted the immediate scan strategy with nested loop join in the implementation of our proposed architecture. 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 made 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 Fig. 4, "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.

B. Merger

In our proposed architecture, 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 Fig. 3. 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 Fig. 6. **STATE0** represents the reset state where necessary hardware initialization operations take place. The details of the other states illustrated in Fig. 6 are described below.

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 set 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

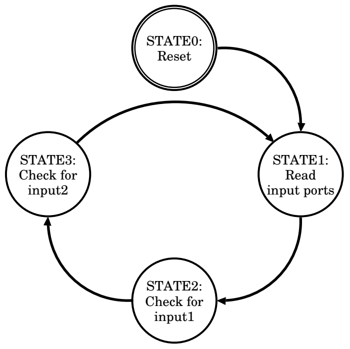


Fig. 6. State transition diagram of the merger.

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.

C. 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 way that the tuples of the input streams R and S flow in opposite direction. However, as shown in Fig. 3, 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 architecture, we suggest a perfect binary tree-like connection for the architecture of result merging network that consists of several merger circuits and their respective connections.

An example of a result-merging network for 16 join cores is illustrated in Fig. 7. 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 perfect binary tree-like connections as illustrated in Fig. 7. As shown at the top of the Fig. 7, we can obtain the final result of the window join operation as a single output stream

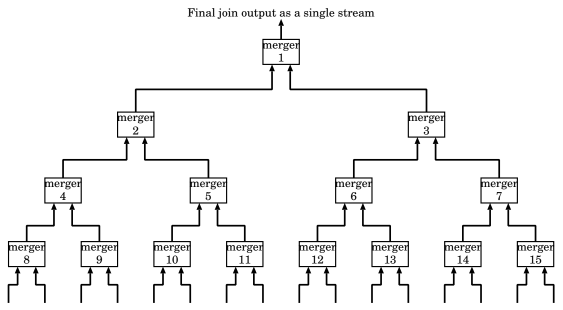


Fig. 7. Perfect binary tree-like connection. An example of a result-merging network for 16 join cores.

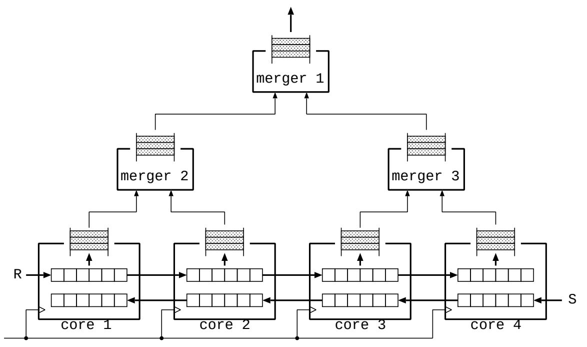


Fig. 8. Connection between join cores and result-merging network.

from output port of the root node i.e. merger 1. 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 the Fig. 7. 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.

For the purpose of clarification, Fig. 8 demonstrates how to connect join cores with corresponding result-merging network. For simplicity, there are only four join cores in Fig. 8; however, the number of join cores and the size of the result-merging network will not affect the approach adopted in Fig. 8.

D. Admission Control

It is a fact that if output rate is above more than the capacity of the output channel, the channel capacity is not enough to transfer all result tuples. In addition, when a large number of result tuples are generated by join cores within a short interval of time, some of the result tuples may be lost due to congestion (buffer overflow) losses. Consequently, the result will be incorrect in either of the cases above mentioned.

In order to avoid the problems with regard to the capacity of the output channel and the limitation of the internal buffer sizes, we have adopted the admission control strategy in our proposed architecture. That is, all of the generated result tuples are transferred to the output channel without any loss by discarding newly arrived tuples to the system when the output rate exceeds the capacity of the channel and/or an internal

TABLE I
SPECIFICATIONS OF XC6VLX240T-1

#. of Slice Registers	301,440
#. of Slice LUTs	150,720
#. of Slices	37,680
#. of BRAM (32KB)	416
#. of DSP48	768

buffer, which is a circular FIFO queue of a join core or a merger circuit, is close to overflow.

As explained previously, each of the FIFO buffer circuits that is used in our implementation has two state flags one of which is full flag that is set to logic 1 when the corresponding buffer is almost full (or completely full). Thus, we can easily grasp the current states of the buffers by observing these flags. Accordingly, the admission control mechanism adopted in our implementation is summarized as follows:

1) If at least one of the full flags of join cores or merger circuits has been set to logic 1, then discard the newly arrived tuples to the system from both of the input streams R and S. Moreover, all of the join cores are suspended until all of the full flags are set to logic 0 in order to refrain from generating new result tuples so that buffer overflow can be avoided.
2) In addition to 1), if any of the full flags of merger circuits has been set to logic 1, then disable the input ports of the corresponding merger circuit until its own full flag is set to logic 0 again.

It should be also indicated that the problem regarding the capacity of the output channel could be resolved by the admission control mechanism described above. For example, in Fig. 8, join cores numbered from 1 to 4 may produce a large number of result tuples within a very short period of time, depending on the characteristics of the input streams. In this case, the output rate will be close to exceed the capacity of the channel; however, it is obvious that when the capacity of the output channel is not enough to transfer all of the result tuples, the FIFO buffer of the "merger1" becomes full, and thus the corresponding full flag is set to logic 1. Consequently, the admission control mechanism will take effect in order to prevent the loss of result tuple.

V. EVALUATION

Our proposed handshake join architecture is implemented as dedicated handshake join hardware on top of a Xilinx FPGA board that is the Virtex-6 FPGA ML605 Evaluation Kit including a Xilinx FPGA XC6VLX240T-1 chip. The specifications of the FPGA, which is used in our implementation, are shown in Table I. During our implementation of handshake join hardware, Xilinx ISE 13.1 Logic Edition is used as an FPGA development environment, as well as XST compiler as a logic synthesis tool.

A. Evaluation Method

Our proposed architecture is implemented as a dedicated hardware for handshake join operation and evaluated as shown in Fig. 9. In addition to handshake join hardware, we have instantiated two buffers, one of which is called "input buffer" while the other is called "output buffer", for the purpose of evaluating the implemented handshake join hardware. Both of these buffer components are instantiated by using the dedicated Block RAM (BRAM) primitives that are available in the Xilinx FPGAs.

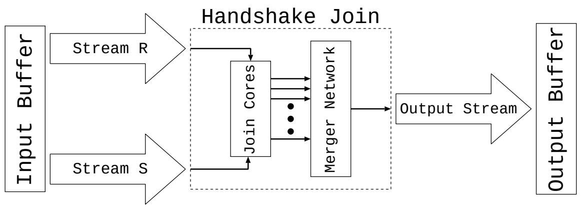


Fig. 9. Evaluation of the implemented hardware for handshake join operation

Before applying the join operation, a number of input tuples of the stream R and S are generated according to a match rate and stored in the input buffer respectively. It should be also noted that these input tuples are placed in a random order.

After preparing the input data, input tuples of the stream R and S are transferred to the handshake join operator and join operation is applied to the input tuples that are accepted by the operator. While processing input tuples, the handshake join operator generates result tuples when join condition is met. Accordingly, the generated result tuples are stored to the output buffer so that all of the result tuples can be transferred as a single output stream without any loss of generated tuples.

B. Resource Usage and Signal Delay

For the purpose of evaluation of our proposed handshake join architecture in this work, we have instantiated different numbers of join cores from 2 up to 64. During the instantiation process, we use the same parameters as in [2] for the tuple structures and size of the windows of each join core. That is, each tuple from both of the input streams R and S consists of 64-bit of data half of which is join key and the remainder is allocated for payload field. Furthermore, each join core produces 96 bit-wide result tuples that are composed of 32-bit join keys and two payload fields, each of which is 32 bit-wide payload. Additionally, we set the window size of each join core to be 8 for each input stream, that is to say each join core keeps up to 8 tuples for each of the input streams R and S.

In Fig. 10, the maximum frequencies of the implemented circuit on a Virtex-6 XC6VLX240T-1 chip are demonstrated for each of the six different configurations. As shown in Fig. 10, all of the maximum operating frequencies are nearly 150MHz, which is the time constraint of 6.67ns (clock period) can be met. At this point, it should be noted that the maximum clock rate at which the implemented circuit can be driven is not declined significantly compared with the original design in FPGAs implementation [2] even though we have added the admission controller and perfect binary tree-like result collection logic in our implementation of proposed architecture.

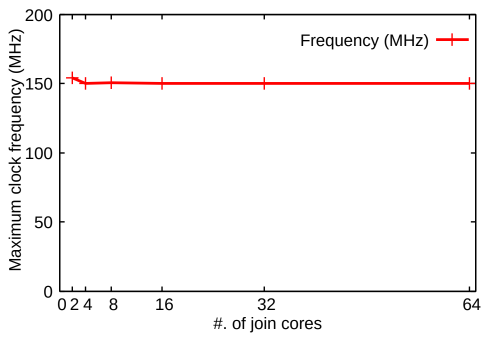


Fig. 10. Maximum frequency of the implemented circuit on a Virtex-6 XC6VLX240T-1 for each of the six different configurations.

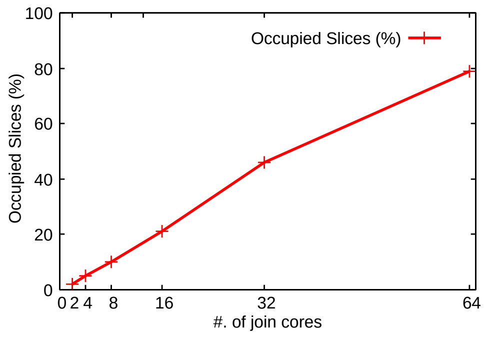


Fig. 11. Overall slice usage of the implemented circuit on a Virtex-6 XC6VLX240T-1for each of the six different configurations.

Likewise, overall slice usage of the implemented circuit is shown in Fig. 11 for each of the six different configurations. The vertical axis of the graph in Fig. 11 represents overall slice usage, that is to say the percentage of the number of occupied slices in our design. As demonstrated in Fig. 11, the overall slice usage is nearly 80% with 64 join cores, and therefore it should be mentioned that 128 join cores could not be instantiated due to shortage of resources on a Virtex-6 XC6VLX240T-1 chip.

C. Performance Evalutaion

The performance of the implemented handshake join architecture is evaluated. In this performance evaluation, the implemented architecture consists of 64 join cores, and it runs at 100MHz. Since our architecture is based on [2], we have adopted the same parameters as in [2] so far. However, the buffer sizes of each join core and merger circuits are not mentioned in [2]. Therefore, we have set the buffer size of each join core and each merger as 8 and 4 tuples, respectively.

In fact, input data for handshake join operator should be streams of tuples generated by real data sources. However, it is difficult to evaluate the performance of the implemented hardware for actual input data because of the variability of the data generated by real data sources. Moreover, buffering of the input streams and transferring input tuples from some different interfaces to the implemented hardware cause additional difficulty in our evaluation. Hence, the performance is

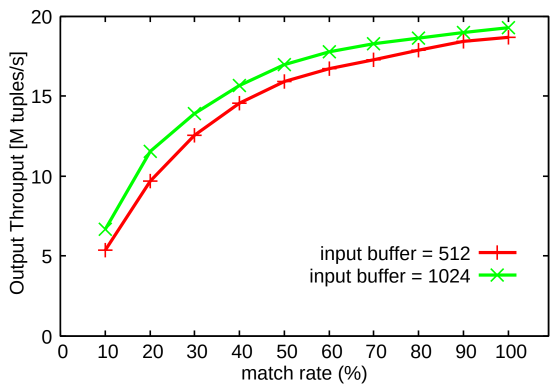


Fig. 12. Output tuples throughput.

evaluated by applying join operation to the data stored into the input buffer of the implemented hardware as illustrated in Fig.9. In this paper, the evaluation is conducted with two different input buffer sizes one of which is 512 tuples (= 1 window size) and the other is 1024 tuples (= 2 window size).

The results of the evaluation of throughput performances are shown in Fig.12 and Fig.13. In Fig.12, the X-axis represents match-rate of input data, and the Y-axis stands for the output throughput of the handshake join hardware when join operation is applied to all of the tuples stored in the input buffer. As shown in Fig.12, the rate of increase in output throughput is lessened while the match-rate of the input tuples is increased. This is because join operation generates a large number of result tuples when the match-rate is raised. Consequently, internal buffers, which are included in join cores and merger circuits, are frequently congested and the implemented admission control mechanism interrupts the input tuples in order to avoid buffer overflow problems. As a result, the time required to apply the join operation to all of the tuples stored in the input buffer increases due to the frequent interruption of the operation.

The input throughput performance is shown in Fig.13. The input throughput is defined as the number of input tuples that can be handled by the join operator without dropping any tuples per a second. In this graph, X-axis is match-rate of input data, and Y-axis is input tuples throughput (M tuples/s). The line labeled "nested-loop join" is the performance estimation of nested-loop join implemented in Terada et al.[7]. The input buffer size of the nested-loop join is also 512 tuples and it also runs at 100MHz for regulating the condition. As illustrated in Fig. 13, owing to the frequent interruption by the admission control mechanism, the input throughput of the handshake join operator becomes lower than nested-loop implementation if the match-rate of the input tuples is increased enough. This result also shows that the implemented join handles higher input throughput than nested-loop join when match-rate is low. Both of the implemented join and nested-loop join interrupts data injection from input buffer when the buffer for the output port is full. However, the admission control mechanism interrupts the entire join cores in the handshake

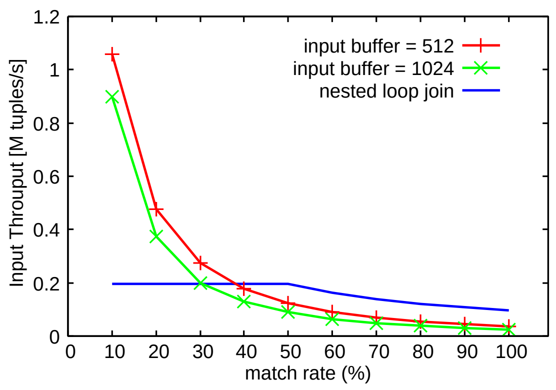


Fig. 13. Input tuples throughput.

join architecture regardless of necessity because their design is based on [2] and they have to operate in a synchronous manner. This causes drawbacks of input throughput of the implemented architecture.

VI. Conclusions and Future Work

In this paper, we proposed an architecture of handshake join for FPGA based on Teubner and Mueller[2]. In handshake join, it is necessary to take into account the problems with regard to the capacity of the output channel and the limitation of the internal buffer sizes, in order to apply join operation to input tuples efficiently in a correct manner. To solve the issues, the merging network and the admission controller are proposed. The proposed additional mechanism contributes correct join operation without any loss.

The proposed architecture is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the operation throughput. The result of maximum clock frequency evaluation shows that the proposed architecture achieves scalability up to 64 cores as mentioned in [2], even though it includes the admission controller and result collection logic. The performance evaluation results show that the proposed architecture handles considerably high rate input stream compared to nested-loop join (implemented in [7]) when match-rate is low.

Future work is as follows. First, to actual applications, there are a lot of cases where the much larger size of windows for join operation are required than the available size of windows in our proposed architecture. In the proposed architecture, since join cores are based on [2], segments of the windows, which are included in join cores, for the input stream are implemented as shift registers by using slices. Therefore, the hardware resource of FPGA limits strictly the available size of window, because of the limitation of the number of slices in a FPGA. In order to increase the window size, some alternative implementation technique to using shift register should be considered.

Secondly, as mentioned in Sec. V-C, the admission controller in the proposed architecture interrupts all of the cores together, synchronously. This causes drawbacks of input throughput of the architecture. For higher performance, we should consider more efficient implementation to exploit asynchronous behavior of the cores.

Finally, we plan to evaluate the proposed architecture for practical application data and determine the suitable size of buffers in mergers and join cores for it.

Acknowledgement

This work is partially supported by “KAKENHI (#22700090)”, “KAKENHI (#23700054)”, and “Early-concept Grants for Exploratory Research on New-generation Network”.

References

[1] J. Kang, J.F. Naughton, and S.D. Viglas. Evaluating window joins over unbounded streams. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 341 – 352, march 2003.

[2] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD ’11, pages 625–636, New York, NY, USA, 2011. ACM.

[3] Buğra Gedik, Rajesh R. Bordawekar, and Philip S. Yu. Celljoin: a parallel stream join operator for the cell processor. *The VLDB Journal*, 18:501–519, April 2009.

[4] J. Teubner, R. Mueller, and G. Alonso. Fpga acceleration for the frequent item problem. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 669 –680, march 2010.

[5] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[6] Takefumi Miyoshi, Hideyuki Kawashima, Yuta Terada, and Tsutomu Yoshinaga. A Coarse Grain Reconfigurable Processor Architecture for Stream Processing Engine. In *21st International Conference on Field Programmable Logic and Applications, 2011. FPL 2011.*, Sep. 2011.

[7] 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.*, volume 110, pages 181–186, 2011.