

Design and Implementation of a Merging Network Architecture for Handshake Join Operator 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, yosinaga@is.uec.ac.jp

† e-trees.Japan, Inc,

Daiwa Building 3F, 2-9-2 Oowada-cho, Hachioji, Tokyo, 192-0045

E-mail: miyoshi@e-trees.jp

‡ University of Tsukuba, Japan

1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577, Japan

E-mail: kawasima@cs.tsukuba.ac.jp

Abstract—A novel merging network architecture is proposed for a handshake join operator in order to achieve much higher data throughput than ever before. Handshake join is a highly parallelized algorithm for window-based stream joins. Result collection performed by a merging network is a significant design issue for the handshake join operator because the merging network becomes an overwhelming bottleneck for scalable performance. To address the issue, an *adaptive merging network* is proposed for hardware implementation of the algorithm. The proposed architecture is implemented on an FPGA and it is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the performance. Experimental results demonstrate up to 16.3 times higher throughput than nested loops-style join implementation without dropping any tuples. To the best of our knowledge, this is the best performance for handshake join operator implemented on an FPGA.

I. INTRODUCTION

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

Most of today's modern relational *database management systems (DBMSs)* offer powerful and sophisticated 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 or *data stream management systems (DSMSs)* also support a similar operation set. One of these basic operations is called stream join or window join [1] that introduces window semantics besides value-based join predicates.

Stream databases deal with unbounded streams of data that have to be processed immediately for real-time applications. Infinite inputs cause a practical problem when a join operation is applied to unbounded input streams. It is stated in [1]

that 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. 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 join operation is evaluated over the windows.

The dominant strategy for executing window-based stream join operation is mostly sequential even though stream joins are fundamental and costly operations in stream databases. The parallelization of stream join operation is quite important to increase the performance. Teubner and Mueller have provided new insight into stream join algorithm, and proposed a new approach, namely *handshake join*. It is a stream join algorithm that can support very high degrees of parallelism and achieve higher throughput rates [2]. They demonstrate a software implementation using a modern multi-core CPU. It considerably outperforms CellJoin [3], which is another well-known implementation of stream join for the Cell processor.

It is mentioned in [2] that the new approach brought by handshake join can naturally take advantage of hardware parallelism. A complete hardware design and an implementation of handshake join are presented in our previous work [4]. The experimental results in [4], however, indicate that the handshake join operator can achieve high throughput performance compared with a conventional approach, proposed by Terada et al. [5], only at low match rates. Consequently, a significant problem regarding the throughput performance has yet to be solved for hardware implementation of handshake join.

A stream join operator can generate a large number of results, depending on the dynamic characteristics of input streams. Moreover, the concurrent execution of multiple join processing units (cores) results in a higher output rate than that of a sequential execution because the same number of results is produced in a shorter time. This causes severe degradation of the throughput performance of the handshake join operator implemented in [4], especially at high match rates.

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. The problem is, therefore, how to design and implement an efficient merging network in order to overcome the degradation of the performance.

The objective of this paper is to address the above problem by proposing an adaptive merging network. An appropriate network model and a careful implementation are extremely important to improve the performance. Accordingly, a markedly different network structure is proposed, overcoming a critical disadvantage of the merging network adopted in [4].

The hardware resource usage and the signal delays are significant factors for the overall design. This paper also intends to clarify these issues. For this purpose, the proposed architecture is implemented as a complete handshake join operator on an FPGA and evaluated as a case study. As a result, the paper presents the maximum clock frequency and the logic size of the implemented handshake join hardware.

In our view, the main contribution of the paper is to propose, design, and implement an adaptive merging network for the handshake join. The handshake join operator with the adaptive merging network substantially outperforms both methods proposed in [4] and [5] under all conditions. The paper presents evaluation results of the throughput comparison with the two implementations of window join operators. To the best of our knowledge, this is the best performance for handshake join operator implemented on an 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 design issues on an FPGA. Section 4 proposes the details of handshake join architecture. Then, Section 5 evaluates the proposed architecture. Finally, Section 6 gives our conclusions and identifies future work.

II. BACKGROUND AND RELATED WORK

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 streams.

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 FPGAs are used for building application-specific hardware [10]–[12].

How to implement window joins is a challenging task because of the tight response-time restriction and heavy computational cost. Efficient implementation method of window joins is required for stream databases in order to meet the time requirement and overcome the heavy computational cost. Consequently, acceleration of the window-based stream join is a significant research issue regarding stream databases.

It is mentioned in [2] that the M3Join proposed by Qian *et al.* [13] implements the join step as a single parallel lookup; however, this approach causes the significant performance drop for larger join windows. Terada *et al.* [5] suggest an implementation of window join on an FPGA. Nevertheless, only two join processes are concurrently executed since the approach adopted in [5] is based on sequential execution. On the other hand, the pipelining approach and the data flow model of handshake join do not suffer from these limitations.

A hardware implementation of handshake join is proposed in [4] to increase the throughput performance by taking advantage of concurrent execution of join units (cores). The handshake join operator implemented in [4] can achieve high throughput rate compared to [5] when the match rate is low. However, the merging network suffers from congestion and it becomes a critical bottleneck for the performance if the match rate is increased. Consequently, the performance is severely degraded when the match rate is high. Details of the handshake join are discussed in the following section.

III. HANDSHAKE JOIN AND ITS RESEARCH ISSUES

A. Handshake Join

The basic idea of the handshake join [2] is to consider two input streams allowed to flow in opposite direction. With this approach, we obtain significant advantages regarding parallelization and scalability. It is stated in [2] 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 given below) 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 [2].

Assuming two input streams (stream R and S) and a newly arrived tuple r from stream R , each step of the three-step procedure [1] is described as follows:

- 1) *Scan* the window for S to find tuples matching r .
- 2) *Insert* the new tuple r into the window for R .
- 3) *Invalidate* all expired tuples in the window for R .

It should be noted that a new tuple arriving from stream S is handled symmetrically.

It is mentioned in [2] that, in general, the three-step procedure corresponds to a *nested loops-style* join evaluation. This 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 data flow-style processing model without a centralized coordinator is proposed with the handshake join approach. It is indicated in [2] that handshake join produces the same outputs

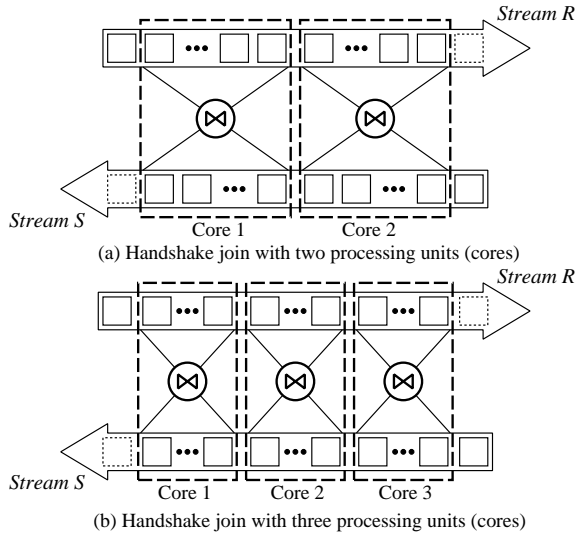


Fig. 1. The parallelization of handshake join.

as classical window join procedure, and it can be regarded as a safe substitute for traditional window join implementations.

The parallelization of handshake join is illustrated in Fig. 1 (adopted from [4]). Each rectangular box represents a tuple from two input streams. As shown in Fig. 1, the degree of parallelism can be easily increased by adding more processing units. All tuple comparisons and evaluation of the join condition are carried out locally and independently since each core is responsible for only its own segment of the two windows. 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.

It should be also noted that each join core only requires local core-to-core communication for transferring tuples of the streams to its adjacent cores. In addition, each core performs the same operation as the other cores in a synchronous manner. From this point of view, join cores and their connections can be regarded as a one-dimensional linear systolic array.

Kung and Leiserson [14] proposed the idea of systolic array that is a structure composed of an array of processors for VLSI implementation. It is stated in [14] 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 very similar to the join arrays [15] proposed for relational databases.

B. Design Issues of Handshake Join

Fig. 2 illustrates the overview of the hardware model of handshake join with 4 join cores, which is the implemented system in [4]. As shown in Fig. 2, join cores are aligned side by side so that the 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 4 sub-windows over 4 join cores respectively.

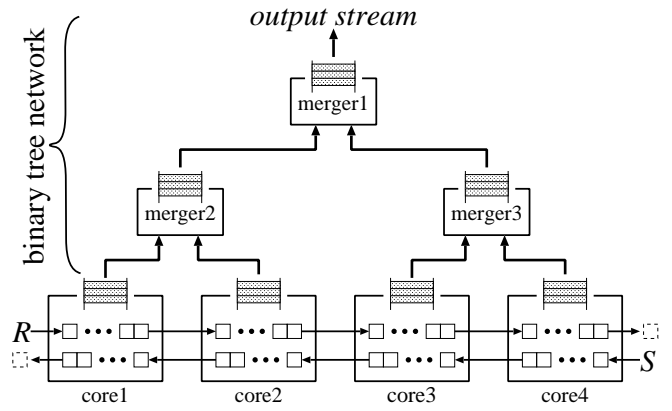



Fig. 2. Binary tree merging network with 4 join cores.

It is obvious that efficient utilization of the massive hardware parallelism is necessary to improve the overall performance of a handshake join operator as a stream join accelerator. Nevertheless, efficient usage of hardware parallelism involves non-trivial aspects. In fact, two main issues have to be considered besides the join cores when it comes to design and implementation of the handshake join hardware:

- 1) a scalable mechanism that collects result tuples and combines them to form a single output stream,
- 2) a lossless flow control mechanism that transfers all results to the output port,

Once these issues are properly addressed, handshake join can become a promising algorithm for stream join accelerators.

First, result collection is an important issue to be solved. For this purpose, a simple binary tree network is proposed as a merging network in [4]. A result merging network is needed to merge all results into a single stream as the final join output. Merging network is a result merging logic that consists of a number of merger units. It should be scalable so as to merge result tuples even if the number of join cores is increased. As illustrated in Fig. 2, a binary tree network is placed on top of the join cores. As shown at the top of Fig. 2, results of the join operation are obtained as a single output stream from the output port of the root node, i.e. *merger1*.

Secondly, the limitation of the internal buffer sizes is considered as a critical problem. As shown in Fig. 2, each join core and merger include FIFO buffers (indicated as  in Fig. 2). Some of the result tuples overflow the buffers when the output rate surpasses the *bandwidth* of the merging network and/or the output channel (bandwidth refers to the amount of data transferred per unit time). For example, the bandwidth of the channel is not enough to transfer all of the result tuples if a large number of results are produced by join cores at the same time. In this case, some of the result tuples overflow the buffers and correct results are permanently lost.

To address the issue, an admission control mechanism is proposed in [4]. The mechanism interrupts tuples of the input streams when handshake join operator is unable to handle new input tuples owing to the relatively high input rate of the input

streams. All of the result tuples are transferred to the output channel by rejecting newly arrived tuples to the system when the output rate exceeds the bandwidth of the channel, and/or an internal FIFO buffer of a join core or a merger is close to overflow. In other words, the admission control provides a lossless flow control mechanism between all join cores and the output port.

IV. DESIGN OF THE PROPOSED ARCHITECTURE

It is indicated in [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. 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 [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 Fig. 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 Fig. 2 is 7, only 3 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 4. Furthermore, the buffer utilization is significantly decreased if 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 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 [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 [2]

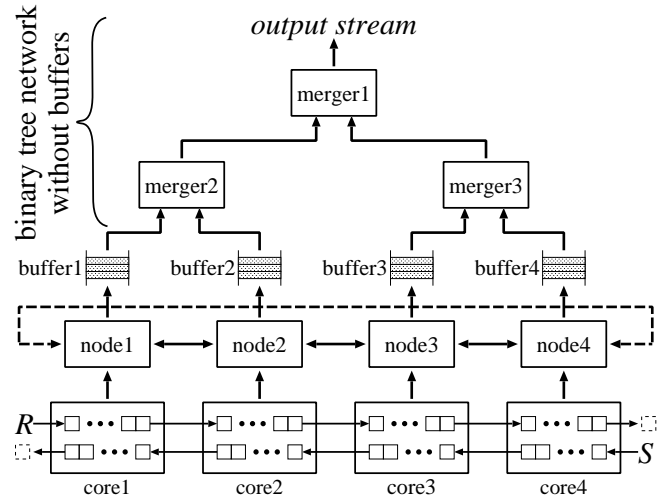


Fig. 3. Adaptive merging network with 4 bufferless join cores.

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 Fig. 3. 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 Fig. 3, 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 Fig. 3), 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 [4], each result tuple can take different paths through the merging network to reach the output port of the root node (*merger1* in Fig. 3).

The problem regarding the hardware architecture of handshake join [4] is discussed and an overview of the architecture of the adaptive merging network is given so far. In the following subsection, the design and implementation of the proposed architecture are described in more detail, especially the difference between the proposed handshake join with the adaptive merging network and one proposed in [4].

A. 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 are one-bit “valid flag” fields 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 (an equi-join is considered as in [4]). 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 [4]. The main difference, however, 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 Fig. 3 instead of storing them in the buffers.

B. Adaptive Merging Network

The adaptive merging network is the most important and notably different part of the handshake join architecture proposed in this paper compared to one proposed in [4]. The simple binary tree network only composed of the mergers is proposed in [4] as the merging network for the handshake join operator. By contrast, a totally different network model is adopted in the present work. 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 Sec. V-B for more details).

1) *Binary tree network*: The binary tree network proposed in [4] (Fig. 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 (Fig. 3) 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.

The difference between the proposed merger and the one proposed in [4] is the existence of the FIFO buffer that stores result tuples. As shown in Fig. 3, each merger forwards the result tuples from its input ports directly to the output port instead of storing them in buffers.

2) *Ring structure and FIFO buffers*: The connections of the ring structure in the proposed merging network is shown in Fig. 4. In proposed design, the bidirectional links are considered as two directed links between each pair of nodes.

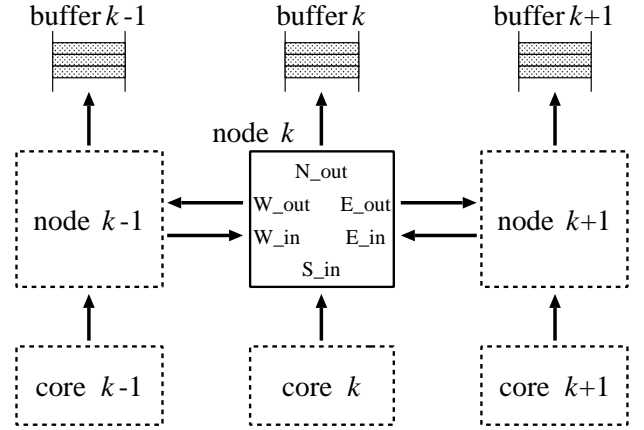


Fig. 4. The connections of the ring structure in the merging network.

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 Fig. 4, *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 lowest distance to the destination of the packet [16]. 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 [16]. 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 [16] 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. 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.

C. Admission Control

The bandwidth of the output channel is not enough to transfer all result tuples when output rate is above more than the 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 [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 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 lossless 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 (set to logic 0).

The above rule guarantees that all of the result tuples generated by join cores will reach the output port of the root node (*merger1* in Fig. 3) in the binary tree network.

V. EVALUATION

The proposed architecture is implemented on a Virtex-6 FPGA ML605 Evaluation Kit including a XC6VLX240T-1 chip. The specification of the FPGA used in the design is given in Table I. Xilinx ISE 13.1 Logic Edition is used as an FPGA development environment.

TABLE I
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

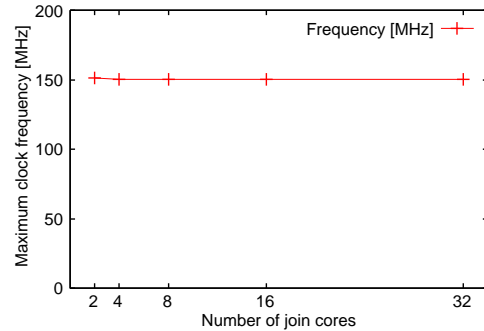


Fig. 5. Maximum clock frequency of the implemented circuit.

A. Resource Usage and Clock Frequency

The hardware resource usage and the clock frequency are evaluated for five different configurations. The different numbers of join cores (2^i where $i = 1, \dots, 5$) and corresponding merging networks are instantiated on the FPGA. The same parameters as [4] is 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 Fig. 5. The x-axis and the y-axis represent the number of join cores and the clock frequency, respectively. As shown in Fig. 5, 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 II. In addition, the percentage of the overall resource consumption is shown in Fig. 6. The y-axis of Fig. 6 represents the percentage of the used resources. As shown in Fig. 6, 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.

Fig. 7 shows the result of the comparison of BRAM utilization between the baseline implementation [4] and the proposed implementation. The y-axis of Fig. 7 represents the total number of BRAMs included in each handshake join operator. As shown in Fig. 7, both of the lines labeled “baseline” and “proposed” linearly increase with the increasing number of join cores. It should be emphasized that the

TABLE II
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

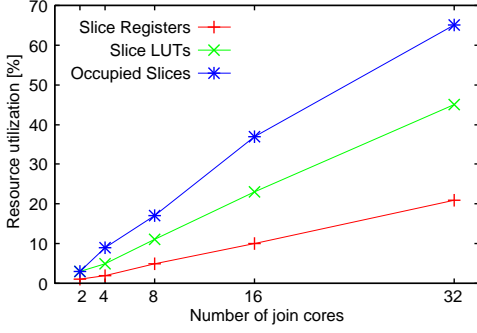


Fig. 6. Overall resource consumption of the implemented circuit.

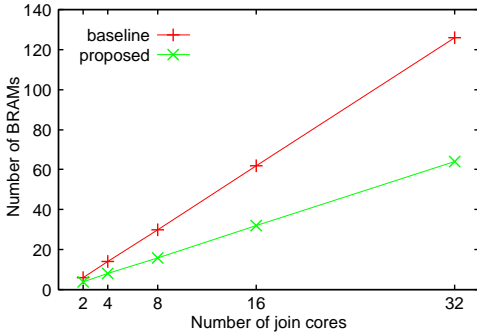


Fig. 7. Comparison of BRAM utilization.

proposed implementation requires fewer BRAMs than the baseline implementation does.

The results of Fig. 5, Fig. 6, and Fig. 7 lead us to the conclusion that the proposed design is scalable in terms of the resource usage and the signal delay.

B. Performance Evaluation

In the performance evaluation, the same evaluation model as [4] is adopted as shown in Fig. 8. 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

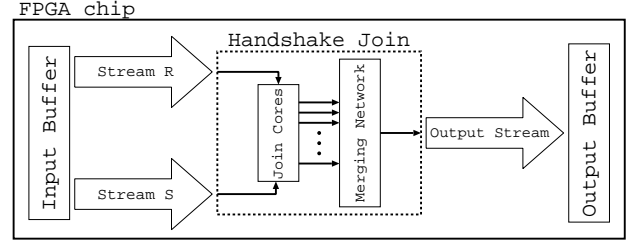


Fig. 8. Evaluation of the handshake join hardware.

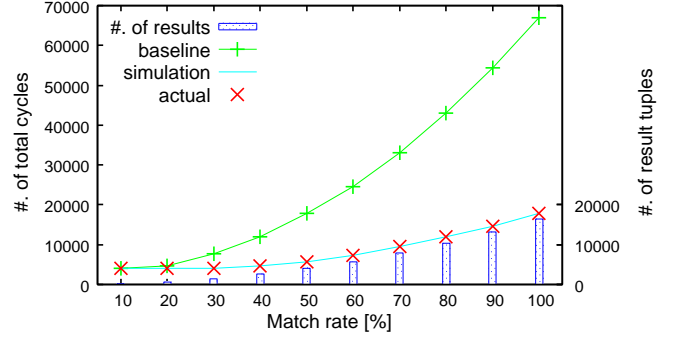


Fig. 9. The number of result tuples and the total number of cycles required to complete the join operation.

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.

Fig. 9 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.

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 implemented in [4]. The same number of result tuples is generated by both of the join operators, and it is indicated as a bar chart in Fig. 9.

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 Fig. 9. 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 Fig. 10. This is the maximum throughput data rate that can be handled by each join operator. 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 [4] and the nested-loop join [5], respectively. It should

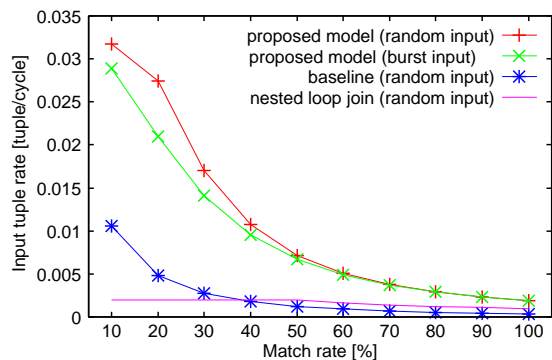


Fig. 10. Throughput performance comparison (64 join cores).

be noted that the size of the input buffer for the nested-loop join 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 Fig. 10.

As shown in the graph, the baseline can achieve higher throughput rate than nested-loop join when the match rate is lower than 40%. On the other hand, the proposed model can achieve far higher throughput rate than nested-loop join even if the match rate is increased. Furthermore, the proposed architecture can handle high input rates compared to others despite burst inputs which can be considered as the worst case. These data lead us to the conclusion that the proposed architecture can considerably outperform both the handshake join [4] and the nested-loop join [5].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proposed an adaptive merging network for handshake join by examining the weakness of the architecture proposed in [4], and presented an implementation of the handshake join operator 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. The suitable network architecture and the careful design are key requirements to alleviate the bottleneck and achieve higher throughput performance.

The performance evaluation results show that the proposed architecture handles considerably high input rate compared with the handshake join [4] and the nested-loop join [5]. In particular, the proposed implementation achieves more than 5.2 times higher throughput compared to the baseline implementation [4] at the highest match rate (*i.e.*, 100%). Furthermore, it also outperforms [5], demonstrating up to 16.3 times higher throughput without dropping any tuples. To

the best of our knowledge, this is the best performance for handshake join operator implemented on an FPGA.

The proposed architecture is also evaluated in terms of the hardware resource usage and the maximum clock frequency. The results of evaluation show that the proposed architecture achieves scalability up to 32 join cores as mentioned in [2], even though it includes the adaptive merging network with the admission control logic.

Future work is as follows. First, the size of windows should be increased for practical applications. In the proposed implementation, each window of input streams is implemented as a large shift register since the design of join cores are based on [4]. In order to increase the window size, an alternative implementation technique should be considered. Secondly, we plan to evaluate the proposed architecture for practical application and determine the suitable size of buffers in the merging network.

ACKNOWLEDGMENT

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. Viglas, “Evaluating window joins over unbounded streams,” in *ICDE*, 2003, pp. 341–352.
- [2] J. Teubner and R. Müller, “How soccer players would do stream joins,” in *SIGMOD Conference*, 2011, pp. 625–636.
- [3] B. Gedik, R. Bordawekar, and P. S. Yu, “Celljoin: a parallel stream join operator for the cell processor,” *Vldb J.*, vol. 18, no. 2, pp. 501–519, 2009.
- [4] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, “An implementation of handshake join on FPGA,” in *ICNC*, 2011, pp. 95–104.
- [5] Y. Terada, T. Miyoshi, H. Kawashima, and T. Yoshinaga, “A consideration of window join operator over data streams by using FPGA (in Japanese),” in *IEICE Tech. Rep.*, 2011, pp. 181–186.
- [6] R. Müller, J. Teubner, and G. Alonso, “Data processing on FPGAs,” *PVLDB*, vol. 2, no. 1, pp. 910–921, 2009.
- [7] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh, “Efficient event processing through reconfigurable hardware for algorithmic trading,” *PVLDB*, vol. 3, no. 2, pp. 1525–1528, 2010.
- [8] M. Sadoghi, H. Singh, and H.-A. Jacobsen, “Towards highly parallel event processing through reconfigurable hardware,” in *DaMoN*, 2011, pp. 27–32.
- [9] M. Sadoghi, R. Javed, N. Tarafdar, H. Singh, R. Palaniappan, and H. Jacobsen, “Multi-query stream processing on FPGAs,” in *ICDE*, 2012.
- [10] J. Teubner, R. Müller, and G. Alonso, “Frequent item computation on a chip,” *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 8, pp. 1169–1181, 2011.
- [11] R. Müller, J. Teubner, and G. Alonso, “Streams on wires - a query compiler for FPGAs,” *PVLDB*, vol. 2, no. 1, pp. 229–240, 2009.
- [12] T. Miyoshi, H. Kawashima, Y. Terada, and T. Yoshinaga, “A coarse grain reconfigurable processor architecture for stream processing engine,” in *FPL*, 2011, pp. 490–495.
- [13] J. bo Qian, H. bing Xu, Y. Dong, X. jun Liu, and Y. li Wang, “FPGA acceleration window joins over multiple data streams,” *Journal of Circuits, Systems, and Computers*, vol. 14, no. 4, pp. 813–830, 2005.
- [14] H. T. Kung and C. E. Leiserson, “Systolic arrays (for VLSI),” in *Sparse Matrix Proc.* SIAM, 1979, pp. 256–282.
- [15] H. T. Kung and P. L. Lehman, “Systolic (VLSI) arrays for relational database operations,” in *SIGMOD Conference*, 1980, pp. 105–116.
- [16] T. Moscibroda and O. Mutlu, “A case for bufferless routing in on-chip networks,” in *ISCA*, 2009, pp. 196–207.