PAPER *Special Section on Parallel and Distributed Computing and Networking*

Design and Implementation of a Handshake Join Architecture on FPGA

Yasin OGE[†a)], ***Nonmember***, **Takefumi MIYOSHI**[†b)], ***Member***,
Hideyuki KAWASHIMA[††c)], ***Nonmember***, ***and*** **Tsutomu YOSHINAGA**[†d)], ***Member***

SUMMARY A novel design is proposed to implement highly parallel stream join operators on a field-programmable gate array (FPGA), by examining handshake join algorithm for hardware implementation. The proposed design is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the performance. Experimental results indicate that the proposed implementation can handle considerably high input rates, especially at low match rates. Results of simulation conducted to optimize size of buffers included in join and merge units give a new intuition regarding static and adaptive buffer tuning in handshake join.

key words: *FPGA, data stream processing, window join operator, accelerator, handshake join*

1. 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 amount of data with certain time restrictions. Low-latency and high-throughput processing are key requirements of systems that process unbounded and continuous input streams rather than fixed-size stored data sets.

Most of the modern relational database management systems (DBMSs) have been added superfluous features. All of them should provide basic set operations including union, intersection, difference and Cartesian product. Moreover, they support other operations such as join, selection, projection and division. Likewise, stream databases also support similar operations and one of these fundamental operations is called stream join or window join [1] that introduces window semantics besides value-based join predicates.

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

Teubner and Mueller have provided new insight into stream join algorithm, and proposed a novel approach, namely *handshake join*. It is a stream join algorithm that can support very high degrees of parallelism and attain unprecedented success in throughput speed [2]. They demonstrate a software implementation using a modern multi-core CPU. It considerably outperforms CellJoin [3], which is another well-known implementation of window-based join for the Cell processor. They also mention that handshake join can naturally leverage available hardware parallelism even though a complete hardware design of handshake join is not provided in [2].

Handshake join enables us to parallelize the matching process in a very elegant way; however, there is a practical problem of the approach: results of parallel processes should be collected and merged into a single output stream. In addition, the parallel execution of joins can result in a higher output rate than a sequential execution because the same number of results is produced in a shorter time. In other words, a larger number of results can be produced per unit time, and the merging process would be quickly overloaded. This is the case, for example, with such applications as *TCP SYN Flood* detection [4] where a volume of output may be instantaneously generated, depending on the dynamic characteristics of input streams. Following design issues should be taken into account when it comes to implementing handshake join hardware:

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

The objective of this paper is to address the above issues and evaluate a hardware implementation of handshake join architecture. The proposed design is implemented on an FPGA and evaluated as a case study. In our view, the major contribution of the paper is to identify the problems encountered in the design of handshake join hardware. To the

Manuscript received January 5, 2012.
Manuscript revised May 23, 2012.

[†]The authors are with the Graduate School of Information Systems, The University of Electro-Communications, Chofu-shi, 182–8585 Japan.
[††]The author is with the University of Tsukuba, Tsukuba-shi, 305–8577 Japan.
a) E-mail: oge@comp.is.uec.ac.jp
b) E-mail: miyoshi@is.uec.ac.jp
c) E-mail: kawasima@cs.tsukuba.ac.jp
d) E-mail: yosinaga@is.uec.ac.jp

DOI: 10.1587/transinf.E95.D.2919

best of our knowledge, this is the first paper that proposes a complete design of handshake join, implements it as a dedicated hardware on an FPGA device, and indicates buffer tuning for join and merge units. This paper is an extended version of the authors' previous work [5]. The present version shows further analysis regarding buffer-size optimization, and discusses static and adaptive buffer tuning for the proposed design.

The rest of the paper is organized as follows: Sect. 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, Sect. 5 evaluates the proposed design. After that Sect. 6 gives some discussions, and finally, Sect. 7 gives conclusions and identifies future work.

2. 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 FPGA is used as a platform for building application-specific hardware [10]–[12].

How to implement stream joins is a challenging task in 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.* [14] suggest an implementation of a window join operator on an FPGA. Nevertheless, only two join processes are concurrently executed since the approach adopted in [14] 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. Details of the handshake join are discussed in the following section.

3. Design of Handshake Join

3.1 Handshake Join

The basic idea of the handshake join [2] is to consider two input streams which are 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, presented by Kang *et al.* [1], can be 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.

The three-step procedure satisfies the semantics of window-based join, which is described as follows: A window join operator takes two streams (R and S) as inputs and produces output tuples (r, s), where r is from stream R and s is from stream S, such that;

1. r and s satisfy the join predicate,
2. and r is in the window for stream R at the same time that s is in the window for stream S.

It is mentioned in [2] that, in general, the three-step procedure corresponds to a *nested loops-style* join evaluation; however, the nature of the nested loops-style join evaluation makes it difficult to scale up to a large numbers of processing units. In fact, this is the main reason why only two join processes are executed in [14]. To solve the problem, the distributed data flow-style processing model without a dedicated centralized coordinator is proposed with the handshake join approach. It is indicated in [2] that handshake join produces the same output tuples as classical window-based stream join procedure, and it can be regarded as a safe substitute for traditional window join implementations.

The parallelization of the handshake join operation is illustrated in Fig. 1. In this figure, each rectangular box represents a tuple from two input streams. As shown in the figures, by increasing the number of processing units, the degree of parallelism can be easily increased a higher level than ever achieved before. Since each core is responsible for only its own segment of the two stream windows, all tuple comparisons and evaluation of the join condition are carried out locally and independently. Theoretically, it can be readily scaled up in order to support large window sizes, achieve high throughput rates, and/or handle compute intensive functions of the join conditions.

3.2 Design Issues of Handshake Join

Figure 2 (adopted from [2]) illustrates the general overview of the handshake join with tuple-based window. As shown in Fig. 2, join cores are aligned side by side so that tuples

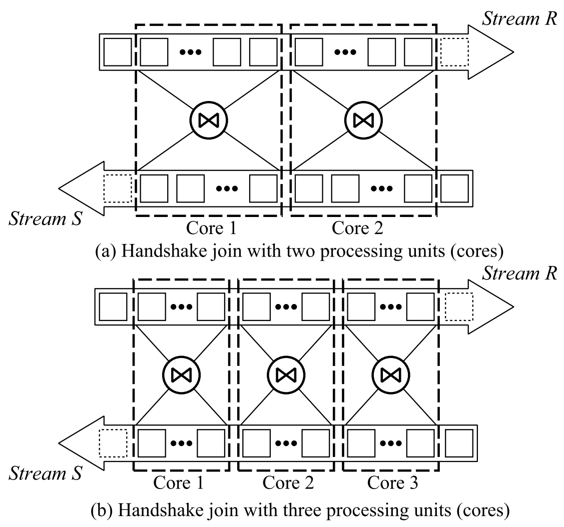

Stream R
Stream S
Core 1
Core 2
(a) Handshake join with two processing units (cores)
Stream R
Stream S
Core 1
Core 2
Core 3
(b) Handshake join with three processing units (cores)

Fig. 1 The parallelization of the handshake join.

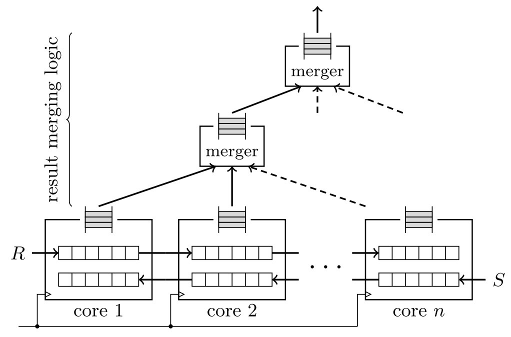


Fig. 2 Overview of the handshake join (adopted from [2]).

of the stream *R* and *S* flow in opposite direction. It can be easily noticed that the windows of the two input streams are divided into *n* sub-windows over *n* join cores. Furthermore, FIFO buffers (indicated as ☰ in the figure) are included in each of the join cores and mergers. Three design issues have to be considered in order to implement handshake join hardware based on Fig. 2.

First, result collection is a main design issue for handshake join hardware. As illustrated in Fig. 2, the result merging logic is placed on top of the join cores. It is not implemented in [2] even though it is stated that a merging network should merge all sub-results generated by each join core.

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

Finally, the limitation of the size of the FIFO buffers is considered as a critical issue. Even if most of the meaningful queries would produce a small amount of results, a possibility of buffer overflow still remains in some applications. For instance, a number of tuples satisfying a join condition can arrive from input streams in *TCP SYN Flood* detection [4]. In fact, it depends on the dynamic characteristics of input streams, particularly whether or not a *TCP SYN Flood* attack [15] occurs. This causes an instantaneous overload of the merging network which leads to the risk of buffer overflow. In this case, some of the results overflow out of the buffers and they are permanently lost. Whether or not the problem would occur really depends on application parameters (*e.g.*, input data rate, match rate, and window size); however, handshake join hardware should be prepared to avoid overflow of the FIFO buffers.

3.3 Design Strategy of Handshake Join

The following components are introduced in the design:

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

Join cores and mergers are shown in Fig. 2. These are fundamental components for join operation and merging results. Merging network is a result merging logic consisting of a number of merger units. It should be scalable to merge result tuples even if the number of join cores is increased. Admission control mechanism provides a flow control between join cores and the output port to prevent data loss due to the buffer overflow. Moreover, the mechanism rejects input tuples when it is difficult to handle high-rate streams causing an overload. It is designed in a way that the proposed approach can be suitably integrated with a load shedding scheme.

It should be also emphasized that join cores only require local core-to-core or core-to-merger communication for data transferring. They concurrently perform the same task in a synchronous manner. From this point of view, join cores can be regarded as a one-dimensional *systolic array*. Kung and Leiserson [16] proposed the idea of systolic array that is a structure composed of an array of processors for VLSI implementation. It is stated in [16] 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 that of the join arrays [17] proposed for relational databases. On the other hand, the proposed design is composed of not only join cores but also the merging network and the admission control mechanism. This is the main difference between a traditional simple systolic array and the proposed design.

4. Architecture of Handshake Join

4.1 Join Core

Join cores evaluate the join condition over the tuples in the windows and generate output tuples. Each segment of the windows is implemented as a shift register. In addition, there are one-bit *valid flag* fields for each tuple in the windows indicating whether the corresponding tuple field is valid or not. Besides the shift registers, it is necessary to implement an output buffer that stores the result tuples.

A circular FIFO queue is implemented as a buffer in each join core. The implementation of the FIFO buffer is based on a dedicated Block RAM (BRAM) primitive that are readily available in FPGAs. Each FIFO buffer has two address registers: *read-address* register and *write-address* register. In addition, two state flags, namely *empty* and *full*, are included in a FIFO buffer. Although the address registers and the state flags would seem self-explanatory, one point should be noted that full flags are asserted whenever corresponding buffers become full or almost full (*i.e.*, there are only few locations left).

When a new tuple comes to a join core, it shifts its own segment of the window one-step to the side. After that, the key value of the received tuple is compared with key values of all tuples in another segment of the window.

4.2 Merger

In the proposed design, a simple structure is adopted for merger units. Each merger has two input and one output ports so as to merge two streams into a single stream. All mergers share a common clock signal with join cores and each of them includes a FIFO buffer, two input-buffer registers, and corresponding flags indicating whether or not data contained in the registers is valid. Result tuples arriving at the input ports are stored to the input-buffer registers, and the corresponding flag is *asserted* (set to logic 1). After loading the data from the input ports, the data stored in each register transferred to the FIFO buffer one by one only if the flags are asserted.

4.3 Merging Network

The proposed design adopts a binary tree network as the merging network of handshake join hardware. Figure 3 demonstrates how to connect join cores with the corresponding merging network. Notice that the number of join cores and the size of the merging network will not affect the approach adopted in Fig. 3. As shown at the top of the figure, the results of the join operation are obtained as a single output stream from output port of the root node, i.e. *merger1*.

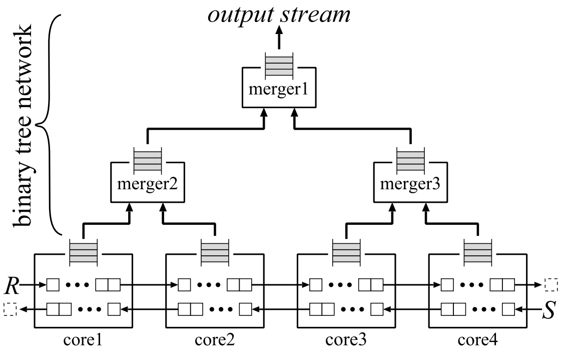


Fig. 3 Connection between join cores and merging network.

4.4 Admission Control

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

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

1. If a full flag is asserted, newly arrived tuples are rejected, and all join cores are suspended until all of the full flags are *de-asserted* (set to logic 0).
2. Furthermore, if any full flag of mergers is asserted, input ports of the corresponding merger are disabled until its own full flag is de-asserted again.

The overhead of the admission control is as follows. All full flags are *ANDed* together, and the result is stored in a flip-flop. In addition, the output of the flip-flop is connected to each join core. For example, if the number of join core is 4 (as shown in Fig. 3), there are 4 bit signals from the full flags of join cores and 3 bit signals from the full flags of mergers. A total of 7 bit signals are *ANDed* together, and the result is stored in a one-bit flip-flop. This one-bit of information indicates whether or not all of the full flags are de-asserted. With the one-bit signal connected to 4 join cores, each of the join cores can determine whether to suspend the matching process.

Notice that the problem regarding the bandwidth of the output channel could be resolved by the admission control. For example, in Fig. 3, the FIFO buffer of the *merger1* becomes full when the bandwidth of the output channel is not enough to transfer all results, and the corresponding full flag is asserted. Consequently, the admission control mechanism takes effect in order to prevent loss of the results due to buffer overflow.

What the admission control provides is the flow control between each join core and the output channel. With the admission control mechanism, the proposed handshake join operator takes responsibility for input tuples accepted by join cores. This means that all results derived from the accepted tuples are transferred to the output channel. In other words, no data loss occurs between each join core and the output channel. On the other hand, this does not always prevent loss of actual join results. The loss of the results can occur when the join operator could not keep up with a high input data rate. It is the fact that a lossless flow of all join results is impossible in such cases since some of the input tuples would be rejected (because of the admission policy). It is stated, however, in [2] that load shedding [18] or distribution [19] can be used if handshake join alone is not sufficient to sustain load. The admission control is consistent with load shedding techniques even though implementation of such a mechanism is out of scope of the paper. The handshake join operator can produce more valuable results once a load shedding mechanism reduces the load of the system because what the admission control guarantees is the join results of the input tuples accepted by the join operator.

5. Evaluation

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

5.1 Resource Usage and Signal Delay

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

The maximum clock frequency of the prototype system is shown in Fig. 4. The x-axis and the y-axis represent the number of join cores and the clock frequency, respectively. As shown in Fig. 4, the graph is almost constant at 150 MHz and the frequency is not declined with increased number of join cores.

The hardware resource usage is given in Fig. 5. In this graph, the y-axis represents the percentage of the number of occupied slices. As shown in the figure, the graph is almost linear, and it can be understood that up to 64 join cores can be instantiated on the FPGA.

The results of the Figs. 4 and 5 lead us to the conclusion that the proposed design is scalable in terms of the resource usage and the signal delay.

5.2 Performance Evalutaion

A simple evaluation model can be used as shown in Fig. 6 to evaluate the throughput performance of the architecture. A number of input tuples are generated according to match rates and stored in the input buffer in a random order (according to a uniform distribution). After that, input tuples are transferred to the handshake join operator. While processing the input tuples, it generates result tuples, and they are stored to the output buffer.

The following parameters are used in the evaluation. The handshake join operator includes 64 join cores, and it runs at 100 MHz. The size of the input buffer is set to 512 tuples, which is the same as the total size of the window. The sizes of the FIFO buffers included in each join core and each merger are set to 8 and 4 tuples, respectively. It should be noted that all results generated by join cores are transferred to the output buffer owing to the admission control. This is confirmed by counting the number of results stored in the output buffer. It is showed that the admission control can work properly (no overflow occurs) even if the sizes of the FIFO buffers are set to such a small value.

The throughput performance is shown in Fig. 7. The line labeled *nested loop join* is the performance estimation of nested loops-style join implemented in [14]. The same parameters as handshake join are used for performance comparison: the size of the input buffer is 512 tuples and it also runs at 100 MHz. The y-axis of Fig. 7 represents the max-

Table 1 Specifications of XC6VLX240T-1.

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

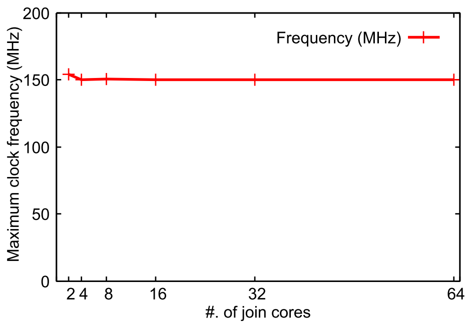


Fig. 4 Maximum clock frequency.

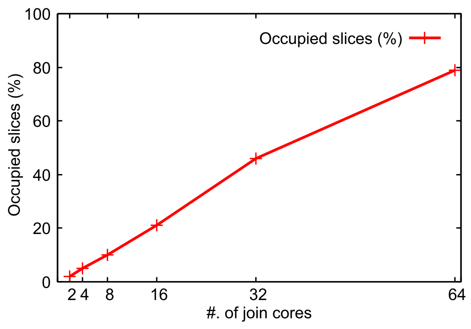


Fig. 5 Overall slice usage.

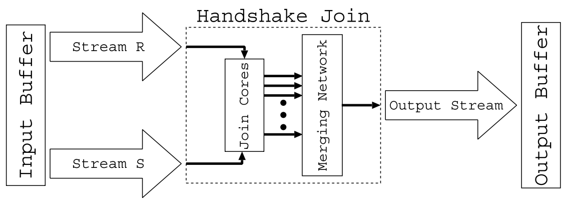


Fig. 6 Evaluation of the handshake join architecture.

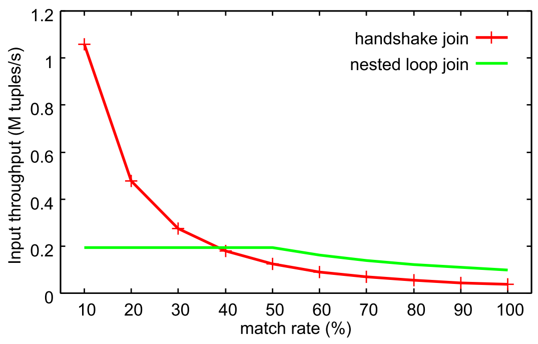


Fig. 7 Maximum input throughput.

imum throughput of input streams that can be handled by each join operator without dropping any input tuple.

Three critical points where an overload can occur should be considered to understand Fig. 7. The first point is between input ports and join cores. As shown in Fig. 6, two input streams (*R* and *S*) flow into the join cores. The second point is between the join cores and the merging network where each join core transfers its sub-results to the merging network. Finally, the third point is between the merging network and the output port. It is the fact that the throughput of the join cores should not depend on match rates. On the other hand, the output rate of the join cores does vary depending on match rates even if the throughput of the input streams remains the same. Furthermore, the parallel execution of joins results in a high output rate because a larger number of results can be produced per unit time (compared to nested loops-style join evaluation) even though the total number of results is not affected by the execution method. With increasing match rates, the join cores produce a considerable number of results, and therefore the second point tends to overload. In addition, the bandwidth of the output channel strictly limits the throughput of the merging network, and this causes an overload in the third point. In fact, what determines the throughput of the entire system is not the join cores but the merging network, especially at a high match rate. This is because the merging network becomes a critical bottleneck as match rate increases.

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

6. Discussion on Buffer Size Tuning

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

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

The same parameters as in Sect. 5.2 are used in the simulation except for the FIFO buffers. The size of the input buffer is 512 tuples, and there are 64 join cores one of which can store up to 8 tuples for each stream. Input and result tuples are 64-bit and 96-bit wide, respectively.

6.1 Static Tuning

As shown in Table 1, there are 416 BRAMs each of which can store up to 36 Kbit data in XC6VLX240T-1 chip. That is to say, we can allocate up to 2^{11} tuples for each join core and merger when BRAM resources are equally allocated among all of the FIFO buffers which are included in join cores and mergers. From this point of view, the total number of cycles required for completion of the join operation is evaluated for different buffer sizes (2^i *where* $i = 2, \ldots, 11$). The simulations are performed at the 100% match rate. According to the results, the numbers of cycles required for the completion of handshake join are 1464443, 1054029 and 1054012 when the buffer sizes are 2^2, 2^3 and 2^4 respectively. Results indicate that the total number of cycles remains the same when the buffer size of each node is equal to or more than 2^4 tuples. What determines the total throughput of the system is not the join cores but the merging network at a high match rate. This is because the merging network becomes a critical bottleneck as match rate increases. Inputs and the output of the merging network are critical points, which can become a bottleneck for the overall system performance. The connection point between the join cores and the merging network becomes a major bottleneck when the buffer size of each node is less than 2^4 tuples. It is possible, however, to alleviate the bottleneck by increasing the buffer size up to 2^4 tuples. Once it has reached 2^4 tuples, the main bottleneck is shifted to the output of the merging network since the bandwidth of the output channel strictly limits the throughput of the merging network. As a result, the increased buffer size no longer alleviates the bottleneck; thus, the total number of cycles remains the same when the buffer size is equal to or more than 2^4 tuples.

So far, the simulation model assumes the same sizes for each buffer of join core and merger. In other words, BRAM resources are uniformly distributed among all FIFO buffers.

Table 2 Buffer size configurations.

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

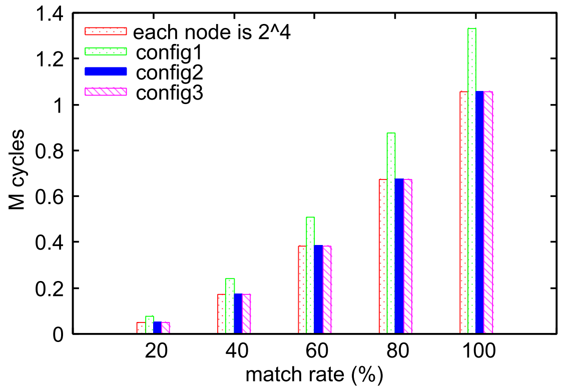


Fig. 8 Results of the simulation for different configurations.

As the next step, the total number of cycles for non-uniform configuration is evaluated. The details of the 3 different configurations are given in Table 2. The first column represents level of the tree. Here, the depth of a node is defined as the length of the path from the root to the node. As a special case, the depth of the root node is 0. The set of all nodes at a given depth is called level of the tree. In these configurations, the buffer size of all nodes at the same depth is equal, and each row of the Table 2 corresponds to the size of each buffer in the same level.

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

The results of the cycle-accurate simulation are shown in Fig. 8. Results indicate that the buffer allocation methods may have great impact on the performance of the handshake join architecture. It is predictable from these results that the buffer sizes of nodes closer to the root should be relatively larger than other nodes located in deeper levels so as to utilize the limited resources efficiently.

6.2 Adaptive Tuning

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

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

In this simulation, the architecture is evaluated with input streams of 3 different characteristics so as to investigate the relation between the characteristic of the input streams and the number of tuples inserted into each buffer. Input tuples which satisfy the join condition are located in the input buffer as follows:

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

The results of the cycle-accurate simulation are shown in Fig. 9. In each graph, the x-axis represents the cycles, and the y-axis stands for the number of tuples stored in the buffer at each cycle. Each graph in Fig. 9 corresponds to the nodes at depth 3 in the binary tree (merging network). Results indicate that the number of tuples stored in the buffer differs from each other.

These data lead us to the conclusion that the adaptive buffer tuning can be applied to the architecture because sufficient space is available in some buffers when some of the others store a relatively large number of tuples. These observations imply that some load-balancing methods such as Dynamically Allocated Multi-Queue Buffers [20] can be used for the purpose of adaptive tuning.

7. Conclusions and Future Work

In this paper, a complete design and implementation of handshake join is presented based on [2]. In handshake join, it is necessary to take into account the result merging logic, and the problems with regard to the limitation of the bandwidth of the output channel and the size of the buffers included in join cores and mergers. The three design issues mentioned in the introduction are addressed by the proposed design including the binary tree network and the admission control mechanism. The proposed additional mechanism contributes to solving the buffer overflow problem in the handshake join operator.

The proposed implementation is evaluated in terms of the hardware resource usage, the maximum clock frequency, and the throughput performance. The result shows that the proposed implementation achieves scalability up to 64 cores as mentioned in [2], even though it includes the merging network and the admission control mechanism. The performance evaluation results show that the handshake join handles considerably high input rate compared with nested loops-style join [14] when the match rate is low. Moreover, simulation results indicate a new intuition regarding static and adaptive tuning of the FIFO buffers included in join

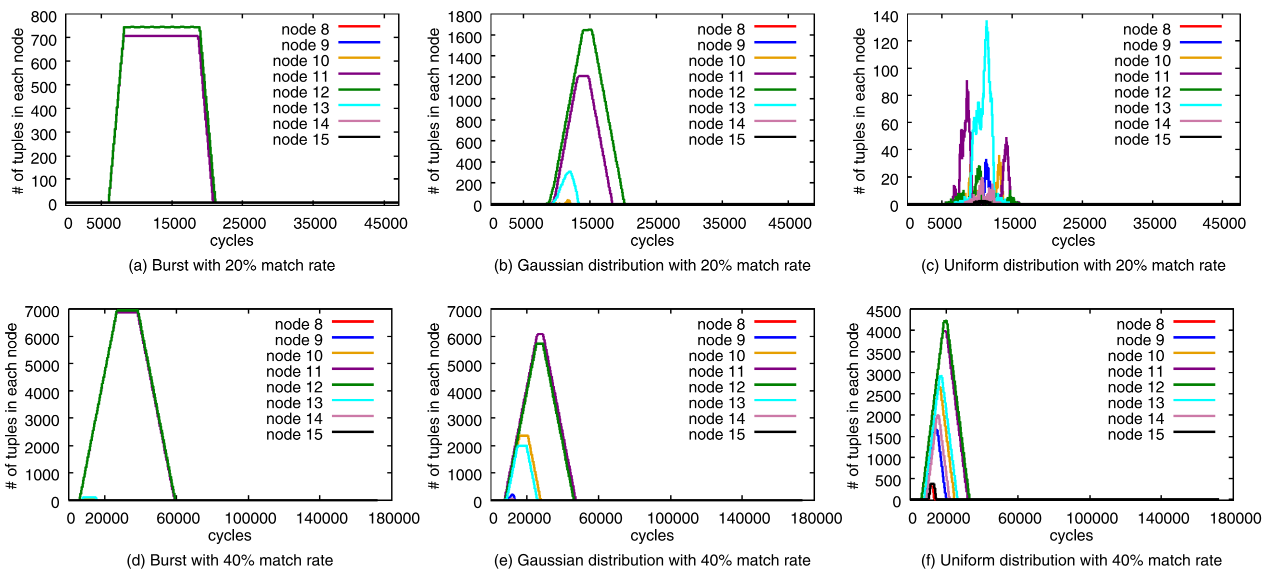


Fig. 9 Results of the simulation for input streams of 3 different characteristics.

cores and mergers.

Future work is as follows. First of all, the current design of join cores will be improved in several aspects. In the proposed design, each segment of the windows for input streams is implemented using shift registers. As a result, the total size of the window is severely limited by the available hardware resources. An alternative implementation technique should be considered to handle large windows. Furthermore, a load balancing strategy for join cores can be implemented to enhance the overall performance. For example, if certain cores are overloaded, the overloaded cores would transfer some of their loads to the neighboring cores. Secondly, the proposed implementation of handshake join tolerates output latency in order to handle higher input rates. The latency, however, is not mainly related to the execution strategy (whether or not join processes are executed in parallel). The longer latency occurs in the merging network after results are produced in each join core; therefore, an improved network structure can offer much better latency characteristics than the proposed one. Finally, the performance of the proposed implementation will be compared to another implementation of window joins (*e.g.*, CellJoin). It should also be evaluated through practical application, determining suitable buffer sizes for join cores and mergers.

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. Viglas, "Evaluating window joins over unbounded streams," ICDE, pp.341–352, 2003.

[2] J. Teubner and R. Mueller, "How soccer players would do stream joins," SIGMOD Conference, pp.625–636, 2011.

[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] T. Johnson, S. Muthukrishnan, O. Spatscheck, and D. Srivastava, "Streams, security and scalability," DBSec, pp.1–15, 2005.

[5] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "An implementation of handshake join on FPGA," ICNC, pp.95–104, 2011.

[6] R. Mueller, 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," DaMoN, pp.27–32, 2011.

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

[10] J. Teubner, R. Mueller, and G. Alonso, "Frequent item computation on a chip," IEEE Trans. Knowl. Data Eng., vol.23, no.8, pp.1169–1181, 2011.

[11] R. Mueller, 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," FPL, pp.490–495, 2011.

[13] J. bo Qian, H. bing Xu, Y. Dong, X. jun Liu, and Y. li Wang, "FPGA acceleration window joins over multiple data streams," J. Circuits Syst. Comput., vol.14, no.4, pp.813–830, 2005.

[14] Y. Terada, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "A consideration of window join operator over data streams by using FPGA," (in Japanese), IEICE Technical Report, RECONF2010-80, 2011.

[15] CERT, "Advisory CA-1996-21 TCP SYN flooding and IP spoofing attacks," 1996.

[16] H.T. Kung and C.E. Leiserson, "Systolic arrays (for VLSI)," Sparse Matrix Proceedings 1978 (Symposium on Sparse Matrix Computations, 1978), pp.256–282, SIAM, 1979.

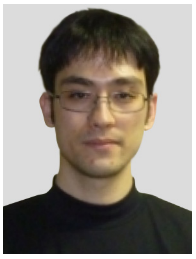
[17] H.T. Kung and P.L. Lehman, "Systolic (VLSI) arrays for relational database operations," SIGMOD Conference, pp.105–116, 1980.

[18] N. Tatbul, U. Çetintemel, S.B. Zdonik, M. Cherniack, and M. Stonebraker, "Load shedding in a data stream manager," VLDB,

pp.309–320, 2003.

[19] D.J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S.B. Zdonik, "The design of the Borealis stream processing engine," CIDR, pp.277–289, 2005.

[20] Y. Tamir and G.L. Frazier, "Dynamically-allocated multi-queue buffers for VLSI communication switches," IEEE Trans. Comput., vol.41, no.6, pp.725–737, 1992.



Yasin Oge received his B.E. degrees (Computer Eng. and Telecommunication Eng.) from Istanbul Technical University, Turkey, in 2010. Since April 2011, he has been studying information network systems at the Graduate School of Information Systems, UEC, Japan.

Takefumi Miyoshi received his B.E., M.E., and D.E. from Tokyo Institute of Technology in 2003, 2005, and 2007, respectively. Since Apr. 2010, he has been with the Graduate School of Information Systems, UEC, where he is an assistant professor. His research interests are compiler techniques, many-core processor architecture, and co-design of hardware and software. He is also a member of ACM, IEEE, and IPSJ.

Hideyuki Kawashima received Ph.D. from Science for Open and Environmental Systems, Graduate School of Keio University, Japan. He was a research associate at Department of Science and Engineering, Keio University from 2005 to 2007. From 2007 to 2011, he was an assistant professor at both Graduate School of Systems and Information Engineering and Center for Computational Sciences, University of Tsukuba, Japan. From 2011, he is an assistant professor at Faculty of Information, Systems and Engineering, University of Tsukuba.

Tsutomu Yoshinaga received his B.E., M.E., and D.E. degrees from Utsunomiya University in 1986, 1988, and 1997, respectively. From 1988 to July 2000, he was a research associate of Faculty of Engineering, Utsunomiya University. He was also a visiting researcher at Electro-Technical Laboratory from 1997 to 1998. Since August 2000, he has been with the Graduate School of Information Systems, UEC, where he is now a professor. His research interests include computer architecture, interconnection networks, and network computing. He is a member of IEEE and IPSJ.