November 5, 2015

RT0969

Computer Science 9 pages

Research Report

A Dynamically Reconfigurable Equi-Joiner on FPGA

Takanori Ueda, Megumi Ito, and Moriyoshi Ohara

IBM Research - Tokyo
IBM Japan, Ltd.
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo 103-8510, Japan

A Dynamically Reconfigurable Equi-Joiner on FPGA

Takanori Ueda, Megumi Ito, Moriyoshi Ohara
IBM Research - Tokyo
Tokyo, Japan
tkueda@jp.ibm.com, megumii@jp.ibm.com, ohara@jp.ibm.com

***Abstract*—As the increase of clock frequencies has slowed, special purpose hardware circuits are becoming increasingly important to accelerate the performance of computing systems. In this context, FPGAs offer advantages over hard-coded ASICs, since FPGAs allow us to use the entire chip to implement optimized algorithms for specific inputs by reconfiguring the circuits at runtime. For example, relational database systems are typically implemented with multiple algorithms for each relational algebraic operation and the one that is expected to process a given query most quickly can be selected as needed. However, previous research on FPGA acceleration for databases has not paid much attention to such algorithm selection. This paper describes an FPGA equi-joiner that switches between two equi-join algorithms, a hash join and a sort-merge join, to fully allocate the FPGA's resources to one algorithm at a time. Our implementation of each algorithm takes advantage of the fact that it can use most of the hardware resources on an FPGA to maximize the size of a key component, a hash table for the hash join and a sort-merge tree for the sort-merge join, which is critical for the join performance. Our experimental results have shown that a simple mathematical model can be used to estimate the execution time of each algorithm for a given data size to select the algorithm appropriately.**

***Keywords*-FPGA; Reconfiguration; Relational Database; Equi Join; Hash Join; Sort Merge Join**

I. Introduction

As the increase of clock frequencies has slowed, special purpose hardware circuits are becoming increasingly important to accelerate the performance of computing systems. FPGAs enable us to build such special purpose hardware with less work because their reconfigurability shortens the turnaround time of the hardware development compared to developing hard-coded ASICs. Along with the ease of development, FPGAs are now attracting developers from various application fields including networking, data compression, and graphic processing. The large number of FPGA applications encourage the mass production of FPGAs, which also drives increases in the circuit density of FPGAs. The developers now can use many of the gates in an FPGA to directly implement applications. In addition, using an FPGA allows us to use the entire chip to implement optimized algorithms for specific inputs by reconfiguring the circuits at runtime, which is another advantage over hard-coded ASICs.

To show the benefits of the runtime reconfiguration, this paper covers an example of exploiting the reconfigurability for database systems. Recent database research has studied methods to use FPGAs in query processing [1]. A query for a database system is described by using a declarative language such as SQL and can typically be processed with predetermined operators, including projection, selection, and join. Thus, the research direction is driven toward how to implement such operators and query processing mechanisms on FPGAs. There are a few proposed FPGA implementations including selection [2], hash join [3], window joins for data streams [4][5][6], and multiple operators [7]. A few papers have also described query processing mechanisms, such as Glacier for continuous queries [8], FQP for event streams [9], and LINQits for accelerating LINQ [10]. These projects address techniques for using FPGAs in query processing.

However, to the best of our knowledge, prior research work has not utilized the FPGA reconfigurability to use the entire chip to implement the fastest algorithm for a specific query. Relational database systems are typically implemented with multiple algorithms for each relational algebraic operation and the one that is expected to process a given query most quickly is selected. FPGAs allow the system to switch among the FPGA implementations depending on the queries, thus exploiting the entire chip for the processing. If we use an ASIC for query processing, we must provide all of the operators on the ASIC, which limits the available resources for each algorithm. Although the FPGA reconfigurability allows the system to exploit the entire chip for query processing, previous research on FPGA acceleration for databases has not paid much attention to such algorithm selection.

A database system that can perform such algorithm selection on an FPGA must have at least these three features:

- Reconfigurable Mechanism
  FPGAs are reconfigurable hardware, but we need additional hardware and software mechanisms to reconfigure the hardware at the time of query processing.
- Relational Algebra Implementations
  If the system has multiple operator implementations, the system can select the one that is expected to process a given query most quickly
- Execution Time Models
  The system needs to estimate the execution time of each algorithm to select the fastest one. Execution time models must be developed for each relational algebra

implementation.

This paper describes an FPGA equi-joiner that switches between two equi-join algorithms, a hash join and a sort-merge join, to fully allocate the FPGA's resources to one algorithm at a time. Our hash join algorithm can use most of the internal memory blocks on an FPGA for a maximum-size hash table, which is critical for the join performance. After it builds hash tables in the memory blocks, it generates the join results by looking-up the tuples with the same keys in the hash table. Similarly, our sort-merge join algorithm can use most of the logic blocks on the FPGA for a maximum-depth comparator tree, which is the key to faster join performance. After sorting with the tree, it merges the sorted tuples to generate the join results. Our joiner selects one of these two algorithms by using a mathematical model to estimate the execution time of each algorithm based on the amount of data.

The remainder of this paper is organized as follows. Section 2 describes the basics of the joins. Section 3 describes the FPGA card that was used for implementing the equi-joiner. Section 4 focuses on our hash join implementation. Section 5 describes our sort-merge join implementation. Section 6 gives a model to predict the execution times of the proposed FPGA equi-join algorithms. The proposed equi-joiner is evaluated in Section 7. Section 8 reviews the related work. We conclude in Section 9.

II. Equi-Join Basics

This section covers the basics of the equi-join operation commonly used in query processing. Equi-join [1] is a binary operation to generate the set of tuples that have matching keys in two tables. In this paper, only one key is considered as the target of the join. More formally, a tuple t is defined as a set consisting of one key k and a set of values $v = \{v_1, v_2, ..., v_V\}$ such that

$$t = (k, v = \{v_1, v_2, ..., v_V\}). \tag{1}$$

Each table is a set of tuples. Here, for tables R and S, the results of the equi-join $R \bowtie S$ are defined as

$$R \bowtie S = \{t_r \cup t_s | t_r \in R \wedge t_s \in S \wedge Eq(t_r, t_s) = 0\} \tag{2}$$

$$Eq(t_r = (k_r, v_r), t_s = (k_s, v_s)) = \begin{cases} 1, & (k_r > k_s) \\ 0, & (k_r = k_s) \\ -1, & (k_r < k_s) \end{cases} \tag{3}$$

The equi-join has to find all of the tuple pairs that have a matching key. Many algorithms and indexing techniques have been proposed to search for such tuples in tables. This paper focuses on hash join and sort-merge join, which are typical algorithms for equi-join.

[1]This paper treats equi-join in the same way as a natural join that combines all of the tuples that have matching keys in two tables.

A. Hash Join

Hash join is a traditional equi-join technique that uses a hash table. We assume there are two tables, R and S. Let $|R|$ and $|S|$ be the numbers of the tuples in R and S respectively. If $|R| \leq |S|$, the hash join performs a join using these three steps:

Step 1) Initialize a hash table in memory. Insert each tuple in R into the hash table. Go to Step 2 when the hash table fills up the entire memory or all of the tuples have been inserted into the hash table.
Step 2) For all of the tuples in S, look for tuples that have a matching key in the hash table. These tuples are the results of the join.
Step 3) If all of the tuples in R were inserted into the hash table, then terminate. Otherwise, return to Step 1 and insert the remaining tuples of R into a fresh hash table.

If the main memory can hold C tuples, then Step 2 has to be executed $\left\lceil \frac{|R|}{C} \right\rceil$ times. This means the number of tuples that this algorithm has to scan is $|R| + \left\lceil \frac{|R|}{C} \right\rceil |S|$. Thus the computational complexity is

$$O(|R||S|) + O(|R|). \tag{4}$$

The hash join completes a join by scanning each table once as long as the memory can hold all of the tuples in R. However, it has to perform multiple scans of S if the memory is smaller than the size of the table. In this case, the execution time increases based on Equation (4).

B. Sort-Merge Join

Sort-merge join is another equi-join algorithm. It first sorts both tables and finds the result tuples by comparing the tuples row by row in the order of the sorted tables. Sort-merge join performs a join using these five steps:

Step 1) Initialize two integer counters $i = 1$ and $j = 1$.
Step 2) Sort the tuples in R and S in ascending key order. The sorted tables are $R = \langle r_1, r_2, ..., r_{|R|} \rangle$ and $S = \langle s_1, s_2, ..., s_{|S|} \rangle$.
Step 3) If $Eq(r_i, s_j) = 0$, then $r_i \cup s_j$ is the join result.
Step 4) If $Eq(r_i, s_j) = 0$, increment both counters. If $Eq(r_i, s_j) = 1$, then increment j. Otherwise, increment i.
Step5) If $i > |R|$ or $j > |S|$, then terminate. Otherwise, return to Step 3.

The sort-merge join involves two sorts and one merge scan. Thus the computational complexity is

$$O(|R|log|R| + |S|log|S|) + O(|R| + |S|). \tag{5}$$

Sort-merge join outperforms hash join when both tables are much larger than the memory and the multiple scans require long time.

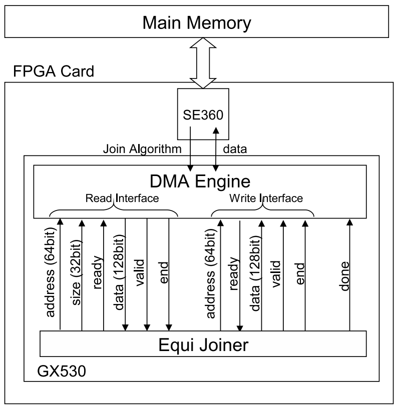

Figure 1. Overview of our FPGA coprocessor card. The card consists of two FPGAs. The DMA engine on the GX530 offers read and write interfaces to the application hardware.

III. FPGA Equi-Joiner

We developed Equi-joiner on our FPGA card that is connected to a computer bus. Figure 1 shows the overall architecture of the card. The card consists of two FPGAs, an Altera Stratix IV SE360 (EP4SE360F35C2) and an Altera Stratix IV GX530 (EP4SGX530KH40C2). The SE360 manages the bus communications. The GX530 has a DMA (Direct Memory Access) engine and contains the circuitry for the running equi-join. With the help of the DMA engine, the equi-joiner can access main memory directly.

A. Algorithm Switch

An Altera Stratix IV has a number of ALMs (Adaptive Logic Modules) [2][11]. However, preparing both join circuits for the FPGA reduces the number of ALMs available for each strategy. This results in a reduction of the size of the hash tables and lower sorting speeds.

Our software allows us to transfer a predefined FPGA join implementation to the GX530 as required. We used Verilog to describe the algorithms. Then we used Quartus II to generate the FPGA images. A Linux tool we developed transfers the generated image file to the GX530 chip. The tool can complete a transfer within a few seconds, which is much faster than the programming tools provided by FPGA manufactures. This feature enables us to quickly switch join implementations based on the optimization decisions that are explained in Section VI.

After the software writes the image of an FPGA implementation, the driver of the join sends two memory addresses for the two tables, their data sizes, and a memory address to store the join results. The FPGA starts the join after it receives this information. When the FPGA finishes its processing, it raises the 'done' signal. The event is sent back to the software.

[2]The number of LEs is converted based on the calculation power of the ALMs because the Stratix IV does not use LEs.

B. Data Transfer Interfaces

The DMA engine exports read and write interfaces to transfer 128-bit data from and to the main memory. As shown in Figure 1, both interfaces use similar signals. The read interface provides 128-bit data that was read from main memory. The engine reads data sequentially from the memory address specified by the 'address' signal for the size specified by the 'size' signal. On each clock edge, the application logic can read 128-bit data when the 'valid' signal is on. When the DMA engine finishes reading the specified amount of data, the 'end' signal is raised. In a similar way, the write interface is for writing 128-bit data into memory. The data is stored sequentially from the memory address specified by the 'address' signal.

Here, the 'ready' signal is important to represent whether the join hardware or DMA engine can receive the next data. In the read interface, the DMA engine must not send the next data until the 'ready' signal from the join hardware is raised. For example, the hash join implementation requires two clocks to process a tuple in an insertion phase and has to cancel the 'ready' signal each time for the first clock cycle of every two clocks. In the write interface, the DMA engine lowers the signal when it cannot write data due to bus contention or any other conditions. With the help of these interfaces, the application hardware can control the data transfers by itself.

C. Memory Blocks

The key components for our hash join implementation are the internal memory blocks. FPGAs have a large number of internal memory blocks, but the size of each memory block is quite small. For example, the Stratix IV GX530 has 1,280 M9K memory blocks and 64 M144K blocks. Table I summarizes the memory block features. Each M9K is able to hold 256 entries of 36-bit data while each M144K is able to hold 4096 entries of 36-bit data. Note that the number of entries that a memory block can store depends on the memory configuration, as shown in Table I.

Figure 2 shows the block diagram of an M9K block when in single-port RAM mode. It can fetch or store one entry in each clock cycle. The location of the fetched or stored data is specified by the address[ ] input. The fetched data is pushed from the q[ ] output. The data to be stored is input into the data[ ] input. This use of memory blocks is a special feature of FPGAs. They form a tightly coupled FPGA architecture of logic cells and memory blocks. The logic cells can access the data in a memory block within each clock cycle.

IV. FPGA Hash Join

This section describes our hash join implementation using the FPGA. A traditional hash table that uses pointer chains is not suitable for handling hash collisions with a FPGA due to its complex operations. The nearly associative memory for FPGAs proposed in [12] is an approximate data structure

Table I
THE MEMORY BLOCK FEATURES OF THE STRATIX IV EP4SGX530

Features	MLABs	M9K Blocks	M144K Blocks
Size (bits)	640	9,216	147,456
The Number of Memory Blocks	(10,624)	1,280	64
Total Size	6,640 Kb	11,520 Kb	9,216 Kb
Available Configurations (depth x width)	64x8, 64x9, 64x10, 32x16, 32x18, 32x20	8Kx1, 4Kx2, 2Kx4, 1Kx8, 1Kx9, 512x16, 512x18, 256x32, 256x36	16Kx8, 16Kx9, 8Kx16, 8Kx18, 4Kx32, 4Kx36, 2Kx64, 2Kx72
Maximum Performance	600 MHz	600 MHz	540 MHz

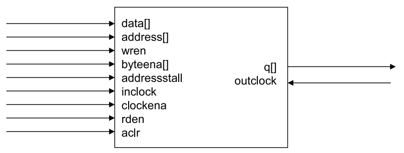

Figure 2. Block Diagram of the M9K Running in Single-Port Mode

and uses a Bloom filter to check for the existence of each key. However, the false positives due to the approximation are usually unacceptable for equi-join.

The implementation in this paper uses our proposed novel hash table for the FPGA to exploit a large number of small-size memory blocks. The key idea is to use different hash functions for each memory block to achieve high efficiency usage of the memory blocks. At first, the implementation inserts tuples of the smaller table into the hash table in the FPGA memory blocks. Then, the tuples of the bigger table are passed through the FPGA and joined with the tuples of smaller table by searching the hash tables. The memory blocks can fetch or store data within a clock cycle. This allows the implementation to exploit the FPGA's high performance.

A. Hash Table on FPGA Memory Blocks

Figure 3 and Figure 4 shows the circuits of our hash table implementation. Our implementation connects the memory blocks into a long chain and uses different hash functions for addressing each memory block. More formally, assume the FPGA has M memory blocks $m_1, m_2, ..., m_M$. The structure divides the blocks into groups $g_1, g_2, ..., g_G$. Let B be the bit width of an element that a memory block can maintain, K be the bit width of a key, and V be the bit width of a value set. Here, each group uses $P = \left\lceil \frac{K+1}{B} \right\rceil$ memory blocks for storing the keys and uses $Q = \left\lceil \frac{V}{B} \right\rceil$ memory blocks for storing the values. Thus, the maximum number of groups is $G = \left\lfloor \frac{M}{P+Q} \right\rfloor$.

Our proposed hash table uses different hash functions $h_1, h_2, ..., h_G$ for each group to determine the address that each tuple is stored to. Each hash function returns an integer value between 1 to C, where C is the depth of the memory blocks. When inserting a tuple $t = (k, v)$ into the table, it finds the smallest i where $g_i[h_i(k)]$ is empty, and then insert t into $g_i[h_i(k)]$. The hash table overflows

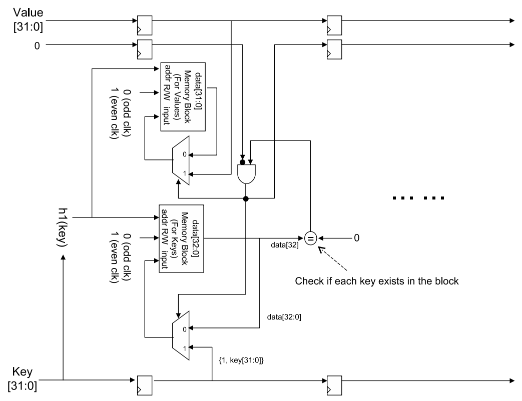

Figure 3. Insertion phase circuits

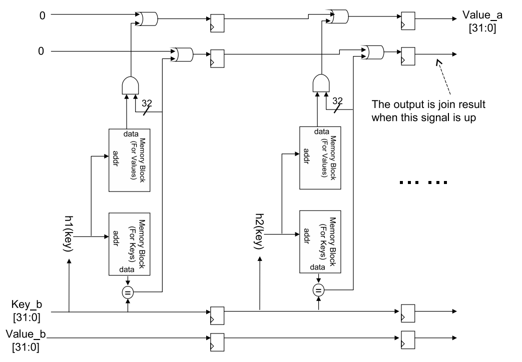

Figure 4. Lookup phase circuits

if the right most memory block $g_G[h_G(k)]$ is not empty when inserting tuple t. To look-up a tuple t, it must check $g_i[h_i(k)]$ for all i. This idea is the same as the multilevel hash table[13][14] though the related paper does not mention any FPGA implementations. Note that the MSB (or LSB) bit of each entry in the memory blocks is used for indicating whether or not the slot is empty.

The space efficiency E of this hash table can be defined as

$$E = \frac{n}{CG}, \tag{6}$$

where n is the number of already inserted tuples when the hash table overflows. The small capacity of each memory block increases the effects of hash contentions with non-uniform distributions of the hash values. Maximizing E is important for the hash tables in the FPGA, because the total size of the memory blocks is limited. Our proposed hash table addresses this problem by using different hash functions, as described above.

B. FPGA Implementation and Operation

Figure 3 shows the abstraction of a group of the hash tables used to insert tuples. It takes two clocks to process each tuple. On the first clock, it checks if the input key exists in the group by using the MSB of the memory block that maintains the keys. On the second clock, if it did not find the key and the tuple was not inserted in the previous group, then it inserts the key and the value at the address determined by the hash function. At the same time, it transfers the tuple to the next group with the information about whether or not the tuple was already inserted.

Figure 4 shows an abstraction of two groups use to look up tuples. It takes one clock to process each tuple. On each clock, the memory blocks output a key and a value on the address determined by the hash function. If the stored key is the same as the input key, then it generates a joined tuple.

The execution procedure of the FPGA hash join is similar to the traditional hash join. Consider performing a join on table R and table S. If $|R| \leq |S|$, tuples in R are passed through from the left to the right of the circuits shown in Figure 3 to build the hash table. After that, the tuples in table S are passed through the circuits shown in Figure 4 to generate the join results by looking up the tuples with the same keys in the hash table. Here are more details about the execution procedures:

Step 1) Initialize an integer $r = 1$.
Step 2) Initialize the hash table on the memory blocks.
Step 3) Start inserting the tuples from the row r of R into the hash table. Go to Step 4 when the hash table overflows or all tuples are inserted into the hash table.
Step 4) For all of the tuples in S, look up the tuples from the hash table that have the same keys. The found tuples are the results of the join.
Step 5) Increment r by the number of inserted tuples of R. Terminate if $r = |R|$. Otherwise, go to Step 2.

V. FPGA Sort-Merge Join

FPGA sort-merge equi-join is the second FPGA equi-join implementation proposed in this paper. It first sorts both tables and then generates the join results by comparing the tuples of both tables in the sorted order. There are several papers about FPGA sorting [15][16][17][18]. Our implementation is similar to an already known implementation [16].

A merge sort tree, the main structure of the implementation, consists of sorter cells that select the smaller value from the two input values. A K-way merge-sort tree can be composed of the sorter cells in the same way as with a binary tree that has K leaf nodes. As an example, Figure 5 shows an 8-way sort merge tree. The sorter cells communicate with

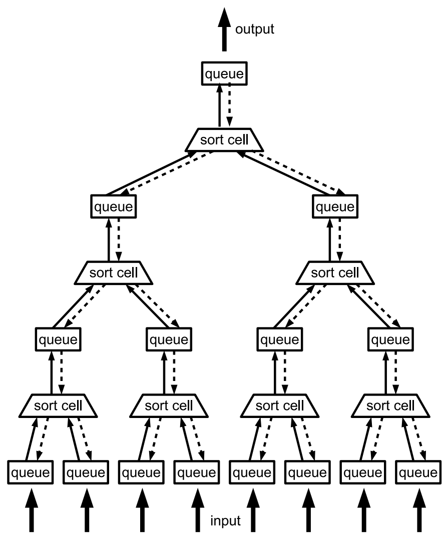

Figure 5. An 8-way merge sorter tree. The tree generates one sorted sequence from the eight sorted sequences. The solid lines represent the directions of tuple transfers. The dashed lines represent the directions of the ready-bit transfers. A sort cell selects the smaller value from the two input values.

the neighboring cells through the FIFO buffers. The buffer transmits a ready bit as long as the FIFO buffer is not full. On each clock edge, the sorter cell sends the smaller value to the FIFO queue if the ready bit is active. As a result, a K-way merge-sort tree generates a sorted sequence from K sorted sequences. The output is sorted in ascending order because all of the sorter cells output the smaller values from their inputs.

Given K sorted sequences $s_1, s_2, ..., s_K$ as input, each invocation of a K-way merge sorter results in a sorted sequence r which consists of all the elements of the inputs $s_1, s_2, ..., s_K$. By invoking the sorter iteratively, we can sort an array $l_1, l_2, ..., l_N$ as described here:

Step 1) Initialize the sequences as $s_i = l_i \ (1 \leq i \leq N)$ and $s_i = \phi \ (i > N)$. That is, we start from N sequences $s_1, s_2, ..., s_N$, each consisting of 1 element. Initialize an integer $b = 1$.
Step 2) Initialize an integer $p = 1$. Update $b = bK$.
Step 3) Invoke the K-way sorter to sort $s_{K(p-1)+1}, s_{K(p-1)+2}, ..., s_{K(p-1)+K}$ to obtain a sorted sequence r_p as output. Update as $p = p + 1$. Repeat this step $\lceil \frac{N}{b} \rceil$ times.
Step 4) Renew s as $s_i = r_i \ (1 \leq i \leq \lceil \frac{N}{b} \rceil)$, $s_i = \phi \ (i > \lceil \frac{N}{b} \rceil)$. Go to step 2 if $b < N$. Otherwise, terminate.

VI. Execution Time Models

Estimating execution time for a join is essential for the equi-joiner. This section gives a model to estimate the completion time of each join of our two implementations.

A. Basic Model of the FPGA Equi-Join

Let p denote a phase within a join operation. Let w_p be the clock cycles required to process a tuple in phase p. If the phase p has to process N_p tuples and f is the clock speed of the FPGA, then the completion time T_p of phase p is

$$T_p = \frac{w_p N_p}{f}. \tag{7}$$

Assume a join implementation imp consists of n phases $p_1, p_2, ..., p_n$ to generate the complete results. In this case, the total execution time T_{imp} to complete the join can be calculated as

$$T_{imp} = \sum_i T_{p_i} = \sum_i \frac{w_{p_i} N_{p_i}}{f}. \tag{8}$$

B. Execution Time Models of FPGA Joins

This section gives the execution time models from analyzing our two FPGA implementations. The hash join outperforms the sort-merge join when one of tables is smaller than the FPGA memory, because it completes the join operation by scanning the two tables only once. However, when the smallest table is larger than the FPGA memory, then the hash join has to scan the larger table multiple times. In that case, the sort-merge join can outperform the hash join.

1) Hash Join Time Model: Our implementation of hash join described in Section IV consists of two phases, p_1 that inserts the tuples into the hash table, and p_2 that searches for the tuples in the hash table. As described in Section II-A, all of the tuples in the smaller table R are scanned in phase p_1. The larger table S must be scanned $\left\lceil \frac{|R|}{C} \right\rceil$ times in phase p_2, where C is the number of tuples that the FPGA can hold. Let E_{hj} denote the constant overhead to start a hash join. With the above analysis, the execution time of our hash join is described as

$$T_{hj} = \frac{1}{f}\left(|R|w_{p_1} + \left\lceil \frac{|R|}{C} \right\rceil |S|w_{p_2}\right) + E_{hj}. \tag{9}$$

Here, in phase p_1, the FPGA receives a tuple every two clocks, because in the first clock the phase checks if the slot of the hash table that the tuple should be stored into is empty, and then in the second clock inserts the tuple into the slot if it is empty. Thus $w_{p_1} = 2$. The looking-up of tuples from the hash table in phase p_2 requires one clock. Thus $w_{p_2} = 1$.

2) Sort-Merge Join Time Model: Our implementation of the sort-merge equi-join consists of three phases, p_1 and p_2 that sort the two input tables, and p_3 that merges the sorted tables to generate the result. The merging phase p_3 has a more complicated characterization. If the current tuples are a joined result, then both of the input streams are advanced. When the number of tuples of the join result is O, then both of the counters are updated O times (phase p_{3a}). Updating only the counter of R happens $|R| - O$ times (phase p_{3b}). Also, updating only the counter for S happens $|S| - O$ times (phase p_{3c}). Thus, the execution time of the sort-merge join is

$$\begin{aligned} T_{smj} = \frac{1}{f} & \left(|R|\lceil \log_K |R| \rceil w_{p_1} + |S|\lceil \log_K |S| \rceil w_{p_2} \right. \\ & \left. + w_{p_{3a}} O + (|R| - O) w_{p_{3b}} + (|S| - O) w_{p_{3c}} \right) \\ & + E_{smj}. \end{aligned} \tag{10}$$

Here, E_{smj} denotes the constant overhead to start a sort-merge join. This paper does not discuss selectivity estimation, but traditional techniques can be used to estimate the number of joined tuples. For that reason, we use $O = 0$ as a pessimistic estimate. The sorting phase can accept a tuple at every clock. Thus $w_{p_1} = 1, w_{p_2} = 1$. When a tuple is a join result, the system has to read the new tuples from the both tables, which takes two clocks. Otherwise, the merging phase can accept a tuple at every clock. Thus $w_{p_{3a}} = 2, w_{p_{3b}} = w_{p_{3c}} = 1$.

Table II
EXPERIMENTAL ENVIRONMENT

CPU	POWER7 3.55GHz
Memory	256GB
OS	Fedora release 18 (3.9.4-200.fc18.ppc64p7)
FPGA Card	GX530 (EP4SGX530KH40C2) and SE360 (EP4SE360F35C2)

VII. EXPERIMENTAL EVALUATION

This section describes our experiments to confirm the benefit of the algorithm selection. Also, this section shows the proposed hash table can store tuples with the high space efficiency. Table II shows the experimental environment used to evaluate the equi-joiner on an actual system. The FPGA card described in Section III is connected to the machine described in Table II.

A. Space Effectiveness of the Hash Tables

The first experiment seeks to show the space efficiency of our hash table implementation. The spece efficiency is critical for the join performance. To evaluate the efficiency of our hash table, the state of the hash tables while inserting tuples was simulated. Our simple simulator compared the number of required memory blocks to store specified numbers of tuples between cases when the blocks use different hash functions and when the blocks use the same hash function.

Figure 6 shows the results when the tuples have random keys. The results show that using different hash functions improved the efficiency. The space efficiency of our implementation with different hash functions reached 99.70%. If the memory blocks used the same hash function for all of the memory blocks, the space efficiency was decreased to 91.83%. The space efficiency was always higher than that of the corresponding implementation that used the same hash function. This experiment shows that using multiple hash functions improves the utilization of the memory blocks. The simulation used other three different distributions of keys: sequential, uniform random, and Zipf random. The results are omitted here because it was similar with the result of gaussian distribution.

B. *Algorithm Switching Benefits*

Table III shows the logic utilization of the hash join implementation. Our hash join uses the FPGA resources linealy with the number of the hash tables. The size of the hash tables is the key factor of the join performance. Sharing the FPGA resources with the sort-merge join, another join algorithm, reduces the table size.

Figure 7 shows the performance of the two hash join implementations and the sort-merge join when joining a 3GB table with the various size tables. One hash-join implementation had 400 hash tables and the other had 200 hash tables. Our FPGA card requires about 1.5 seconds to transfer an FPGA image to the FPGA. Even so, we can receive the benefits from the algorithm switching because there is the differences longer than 1.5 seconds between the two implementations. Also, we can change the algorithm to the sort-merge join when the table size is large.

Table III
LOGIC UTILIZATION OF HASH JOIN

Tables	Utilization (%)
100	22
200	44
300	65
400	86

C. *Execution Time of Equi-Joiner*

The purpose of the final experiment is to assess the equi-joiner performance. The experiment assumes that all of the source data is stored in memory and all of the results are also stored in memory. First, the benchmark driver generated two artificial tables. All of the keys in each table are unique. Then the driver sends the memory addresses of the two generated tables and the address for the join results. The FPGA joiner does the join and sends the result to the specified address in memory. When the join finishes, the FPGA notifies the driver. In this experiment, the execution time is the difference between the time when the memory addresses are sent and the time when the driver receives the completion notification from the FPGA.

Figure 8 shows the execution time of both join implementations. The hash join outperforms the sort-merge join when the tables are small. If the tables become large, the sort-merge join outperforms the hash join. The equi joiner can select the faster algorithm depending on the model. For the estimation models, we used $E_{hj} = 0.0001[s], E_{smj} = 0.005[s]$ The simple mathematical model can be used to estimate the execution time of each algorithm for a given data size to select the algorithm appropriately.

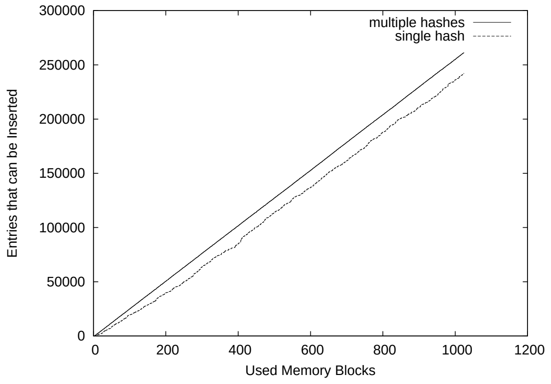

Figure 6. (Gaussian Random Keys) The number of memory blocks used when the specified number of tuples with uniform random keys are inserted.

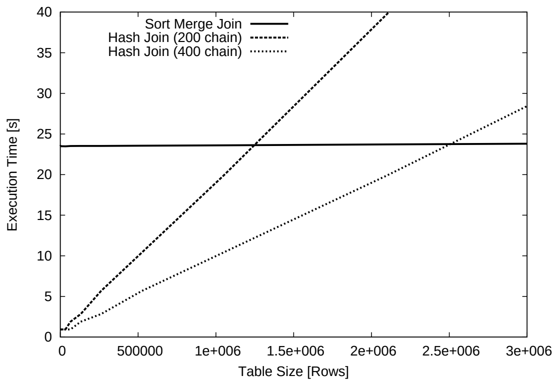

Figure 7. The performance differences between the three algorithms.

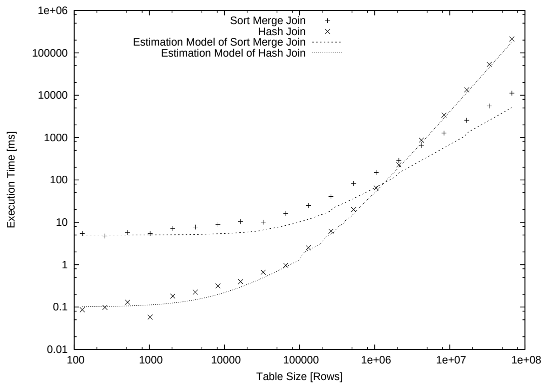

Figure 8. The actual and estimated execution time of both join implementations.

VIII. Related Works

Database researchers only recently started to consider how FPGAs can execute database operations [1]. Neteeza uses FPGAs to filter the data near storage systems [2]. Filtering data that is not required to produce query results can reduce the amount of data transferred from storage to main memory. There is still little literature about FPGA joins [6]. Another paper [7] describes an implementation that supports selection, sorting, and merging to accelerate an entire join operation. A limited number of research papers describe how to apply FPGAs to perform window joins for data streams [4][5]. Mueller et al. developed Glacier, which has a compiler that uses pre-built components to process continuous queries on FPGAs [8]. Najafi et al. also developed a system for event streams [9]. The special feature of their system is that it can accept new query expressions even while it is processing incoming events. These development and research projects are working on a line of using FPGAs for (co-)processing of database systems. There is a paper related to reconfiguration [19]. However, the target operation of the paper is selection and not join. To the best of our knowledge, prior research work has not utilized the FPGA reconfigurability to use the entire chip to implement the fastest algorithm for a specific query.

Regarding data structures, Dhawan et al. proposed near-associative memories using FPGAs [12]. However, this data structure causes false positives, which is unacceptable for the equi-join. Sorting networks allow for high performance sorting in hardware. They have been intensively studied in the past [20]. Sorting networks in FPGAs were well described in [16]. They proposed various connections with comparators to realize bitonic sort, bubble sort, insertion sort, and even-odd merge. However, sorting networks usually require knowing the amount of data in advance. This means they cannot complete the sort operations only in the FPGA when the data is larger than the size the implementation was designed for.

IX. Conclusion

FPGAs allow us to switch among the FPGA implementations depending on the queries, thus exploiting the entire chip for the processing. Although the FPGA reconfigurability can increase the benefits of FPGAs in query processing, previous research on FPGA acceleration for databases has not paid much attention to such algorithm selection.

This paper describes an FPGA equi-joiner that switches between two equi-join algorithms, a hash join and a sort-merge join, to fully allocate the FPGA's resources to one algorithm at a time. Our implementation of each algorithm takes advantage of the fact that it can use most of the hardware resources on an FPGA to maximize the size of a key component, a hash table for the hash join and a sort-merge tree for the sort-merge join, which is critical for the join performance. Our joiner selects one of these two algorithms by using a mathematical model to estimate the execution time of each algorithm based on the amount of data.

Acknowledgment

The authors would like to thank the team in IBM Research - Austin for developing the FPGA card used in this research.

References

[1] R. Mueller and J. Teubner, "Fpga: What's in it for a database?" in *Proc. SIGMOD 2009*.

[2] P. Francisco, "The netezza data appliance architecture: A platform for high performance data warehousing and analytics," *IBM Redbooks*, pp. 1–14, Feb. 2011.

[3] R. J. Halstead, I. Absalyamov, W. A. Najjar, and V. J. Tsotras, "Fpga-based multithreading for in-memory hash joins," ser. CIDR '15, 2015.

[4] Y. Oge, T. Miyoshi, H. Kawashima, and T. Yoshinaga, "An implementation of handshake join on fpga," in *Proc. ICNC 2011*, Nov 2011, pp. 95–104.

[5] J. Teubner and R. Mueller, "How soccer players would do stream joins," in *SIGMOD 2011*.

[6] R. Halstead, B. Sukhwani, H. Min, M. Thoennes, P. Dube, S. Asaad, and B. Iyer, "Accelerating join operation for relational databases with fpgas," in *Proc. FCCM 2013*, pp. 17–20.

[7] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. FPGA 2014*, pp. 151–160.

[8] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires – a query compiler for fpgas," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 229–240, Aug. 2009.

[9] M. Najafi, M. Sadoghi, and H.-A. Jacobsen, "Flexible query processor on fpgas," *Proc. VLDB Endow.*, vol. 6, no. 12, pp. 1310–1313, Aug. 2013.

[10] E. S. Chung, J. D. Davis, and J. Lee, "Linqits: Big data on little clients," in *Proc. ISCA 2013*, 2013, pp. 261–272.

[11] "Stratix iv device handbook volume 1," pp. 1–1–3–24, Sep. 2012. [Online]. Available: http://www.altera.co.jp/literature/hb/stratix-iv/stx4_5v1.pdf

[12] U. Dhawan and A. DeHon, "Area-efficient near-associative memories on fpgas," in *Proc. FPGA 2013*.

[13] A. Kirsch and M. Mitzenmacher, "Simple summaries for hashing with choices," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 218–231, Feb. 2008.

[14] ——, "The power of one move: Hashing schemes for hardware," *IEEE/ACM Trans. Netw.*, vol. 18, no. 6, pp. 1752–1765, Dec. 2010.

[15] D. Koch and J. Torresen, "FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting," in *Proc. FPGA 2011*, Feb. 2011, pp. 45–54.

[16] R. Mueller, J. Teubner, and G. Alonso, “Sorting Networks on FPGAs,” *The VLDB Journal*, vol. 21, no. 1, pp. 1–23, Feb. 2012.

[17] K. Ratnayake and A. Amer, “An FPGA Architecture of Stable-Sorting on a Large Data Volume : Application to Video Signals,” in *Proc. CISS 2007*, Mar. 2007, pp. 431–436.

[18] S. Wong, S. Vassiliadis, and J. Y. Hur, “Parallel Merge Sort on a Binary Tree On-Chip Network,” in *Proc. of ProRISC 2005*, Nov. 2005, pp. 365–368.

[19] C. Dennl, D. Ziener, and J. Teich, “Acceleration of sql restrictions and aggregations through fpga-based dynamic partial reconfiguration,” in *FCCM 2013*, April 2013, pp. 25–28.

[20] D.-L. Lee and K. E. Batcher, “A Multiway Merge Sorting Network,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, no. 2, pp. 211–215, Feb. 1995.