A Scalable, High-Performance Customized Priority Queue

Muhuan Huang*, Kevin Lim+ and Jason Cong*

*University of California, Los Angeles
Computer Science Department
{mhhuang,cong}@cs.ucla.edu

+Hewlett Packard Labs
kevin.lim@hp.com

***Abstract*— Priority queues are abstract data structures where each element is associated with a priority, and the highest priority element is always retrieved first from the queue. The data structure is widely used within databases, including the last stage of a merge-sort, forecasting read-ahead I/O to stream data for the merge-sort, and replacement selection sort. Typical software implementations use a balanced binary-tree based structure, providing O(log N) time for most operations.**

To improve the performance in these cases, we propose several scalable and high-speed FPGA-based implementations of a priority queue. Our insight is that the listed applications primarily use priority queues through "replace" operations, which remove the highest priority element and place a new element into the queue. Thus, our designs are customized for this operation, allowing for a simple and scalable architecture. Our priority queue designs include a register-based array, register-based tree, and BRAM-based tree, which have different benefits and trade-offs of throughput, frequency, and maximum size. Importantly, all designs achieve O(1) time between replace operations.

To incorporate the best aspects of our designs, we propose a Hybrid Priority Queue (H-PQ), which combines a register-based array with multiple BRAM-based trees. This design provides, on average, very fast access times to the top items in the queue (through the register-based array), while scaling to large priority queue sizes (through the BRAM-based trees). In our evaluations, we find that H-PQ achieves 4.3x speedup and 14.3x energy efficiency, compared with the best CPU (Xeon and embedded ARM) implementations.

I. Introduction

In this paper we demonstrate the design of a scalable high-speed FPGA-based priority queue. A priority queue is an abstract data structure which returns elements in order of their associated priority. In database systems, such a data structure is widely used for many purposes: for example, generating the initial sorted runs, merging cache-sized runs in memory, merging disk-based runs and forecasting the most effective read-ahead I/O [1]. It is thus worthwhile to thoroughly profile and optimize the priority queue implementations.

Our FPGA-based priority queue leverages several properties of FPGAs to help improve performance in database systems. FPGAs have massively parallel or fully pipelined on-chip processing units, and thus can achieve a significantly higher processing throughput than CPUs. Furthermore, by offloading tasks, the FPGA frees up CPU cycles to be used instead for other tasks. Compared with ASICs, FPGAs can be configured to meet flexible processing requirements, such as different data types and widths, making them more appealing for actual implementation.

Thanks to recent advantages in high-level synthesis (HLS) tools [2, 3], we are able to design FPGA implementations in a C/C++ based programming flow. HLS greatly reduces programming efforts, allowing us to design and evaluate the performance and scalability of several different priority queue implementations. In this work all the designs are coded in C++.

Priority queues have two basic operations: *dequeue* and *enqueue*. *Dequeue* returns the element that has the highest priority and removes it from the queue; *enqueue* inserts an element, with its specific priority, into the queue. Sometimes a *dequeue-enqueue* operation, or a *replace* operation, is also considered as the third basic operation.

To allow for quick (constant time) identification of the highest-priority element, priority queue implementations usually maintain a partially ordered internal structure, if not fully ordered. Traditional software implementations use a heap to implement a priority queue; a heap stores its internal nodes in a binary tree structure, and maintains the property that any parent node is larger than both of its children nodes. While a heap takes O(1) time to check for the highest-priority element, it takes O(logN) time to insert or remove elements, since upon insertion/removal, it must fix the binary tree to regain the properties of a heap. Thus, tasks that repeatedly insert and remove elements from the priority queue will be bounded by this O(logN) time.

Several hardware priority queue implementations have been proposed in prior work; these are based on either a pipelined heap [4, 5] or a systolic array [6]. These implementations use parallel designs in FPGAs that enable the operations that maintain the data structure properties to occur at the lower levels of the priority queue, while allowing continued operation on the higher level. We will discuss the details of these implementations in Section V. While the concept of a pipelined operation has merit, these approaches have several limitations:

(1) These designs use on-chip memories to store the data

Muhuan Huang worked on this research project while she was a summer intern in Hewlett Packard Labs.

node. Since an on-chip memory access takes at least one clock cycle, many of the pipelined stages — which include fetching data from the on-chip memory, comparing the data and writing data back to memory — end up taking multiple clock cycles. To speed up the implementation, complicated logics and wide-port memories (e.g., 4-entry-wide or 256-bit-wide memories) have to be introduced so as to overlap the successive operations.

(2) The *replace* operation has been largely ignored in prior work, mainly because it tends to use priority queues in network routers. In that case it takes several clock cycles to decode the flow ID out of the dequeued packet and then fetch the next packet from the corresponding flow. Performance-wise a *replace* operation is the same as a *dequeue* followed by a *enqueue*. However, we find the *replace* operation critical when a priority queue is used in a database system. It is favorable to have a single cycle *replace* rather than a multiple-cycle *dequeue-enqueue* operation.

Our work makes the following contributions:

(1) We propose three priority queue implementations based on FPGAs: register-array, register-tree and BRAM-tree. Unlike a traditional priority queue implementation, we aim to **only** support *replace* and *dequeue* operations. By eliminating the *enqueue* operation, we have a simplified but faster design. For example, register-array needs half the storage and half the number of comparators when compared to the systolic array-based implementations. Major priority queue applications in database systems only use *replace* operations.

(2) We evaluate the design trade-offs of the proposed priority queue implementations. The register-array is the best design choice for small priority queues, the size of which is on the order of tens of elements. However, the register-array consumes logic for O(N) comparators which makes it unable to fit on-chip when N is large. The BRAM-tree only consumes logic for O(logN) comparators, but at the cost of slower throughput.

(3) We propose a hybrid priority queue (H-PQ) implementation that combines the best parts of both worlds — register-array and BRAM-tree. Unlike previous BRAM-based pipelined heap implementations, our BRAM-tree design is simple enough that it only needs single-entry-wide ports. We show that H-PQ can support a large-sized priority queue at a reasonably fast throughput.

The remainder of this paper is organized as follows. We give an overview of priority queue applications in database systems and highlight the practical design considerations in Section II. Section III proposes three different priority queue architectures and provides architecture benchmarking. Section IV proposes a hybrid priority queue architecture. Section V discusses the related work. Experimental results and discussions are shown in Section VI. And we conclude in Section VIII.

II. Priority Queue in Database Systems

A. N-way Merge

The priority queue provides a natural way to implement an N-way merge as the last stage of merge-sort, and serves to merge N sorted runs into a single sorted run. The basic operation is to output the smallest element from all the elements not yet output in the N runs.

A priority queue of size N is needed. The priority queue is first fully loaded with the first elements from the N runs. Then the smallest element is output, and is replaced by the next element from the run where it came from.

B. Forecasting Read-Ahead I/O

N-way merge requires proper I/O handling if the initial runs reside in external devices. A standard technique called double-buffering, or ping-pong buffering, can be used to overlap the I/O with computing. Two buffers are maintained for each run, one for burst I/O and the other for computation. The two buffers switch their roles whenever computation on the old data has finished and the I/O buffer has been filled with new data. There are 2N buffers in total.

Double-buffering leads to a low memory utilization since only half the buffers are actually in use for computation. To improve the memory utilization, we can leverage a forecasting technique that only requires N+1 buffers; the one extra buffer is used for I/O prefetching. The key insight is that the next buffer to be emptied is the one whose last item is the smallest. Thus, we can effectively predict which sorted runs to prefetch by examining the last items of all the runs.

To implement such a forecasting technique, a priority queue can be used to organize the last items from the current buffers. It selects the smallest item among them, and reads the next block of items from the corresponding run into the pre-fetch buffer. Then the smallest item is replaced with the last item which was newly inserted into pre-fetch buffer, and then the smallest item is updated, etc.

In the case of FPGAs, the initial runs can be stored in off-chip DRAM or disks (if a disk device is attached to FPGA via SATA). On-chip block memories (BRAMs) are used to buffer the data. Given the limited amount of BRAMs, which is typically several megabytes, it is favorable to forecast the read-ahead I/Os rather than to keep a double buffer for each run.

C. Replacement Selection Sort

Replacement selection sort, also known as tournament sort, is one of the most commonly used run generation techniques for external sorting [1, 7]. Unlike other sorting methods where data are first loaded into memory through burst I/O before sorting actually starts, replacement selection sort is able to sort data in a streamed fashion that overlaps I/O with sorting. More importantly, replacement selection sort generates runs that are twice the size of the memory

on average, thus reducing the number of runs to be merged at the merge phase and reducing the chances to perform multiple merge phase. (If the number of runs is larger than the number of fan-ins of the merge phase, multiple merge phases are needed.)

The replacement selection performs the in-memory sort by passing the data through a large priority queue. It works as follows [8]. (1) Choose as large a priority queue as possible, say of N elements. (2) Load the first N data into the priority queue. (3) Output the smallest data from the priority queue. Read a data from the input and replace the smallest data with the new data. If the new data is smaller than the last one output, it cannot become part of the current run. Mark it as belonging to the next run, treat it as greater than all the unmarked elements in the queue. (4) Repeat (3). When a marked element reaches the top of the queue, terminate the current run and start a new run.

Note that the marked elements need to be reconstituted before a new run. To avoid such overhead and keep the data stream from stalling, the priority queue also needs to maintain the order of the data that are marked as next-run.

D. Putting it All Together

With the above applications in mind, we "prioritize" the practical design considerations as follows.

(1) The priority queues in these applications primarily need replace operations. Enqueue operations only occur at the initialization stage. Thus enqueue operations can be implemented using the replace operations. To initialize the priority queue with the replace operation, the priority queue can be designed to have all entries initialized to $+\infty$ ($-\infty$) for the max (min) priority queue. Then the replace operation can be used to fill up the queue; $+\infty$ ($-\infty$) will always be shifted to the top of the queue, guaranteeing that the replace operations will remove only those special values. Similarly, dequeue can be implemented as a replacement of the top element with $-\infty$ ($+\infty$) for a max (min) priority queue.

(2) The size of priority queues needed in database systems can be very large. As pointed out, replacement selection sort prefers a priority queue that is as large as possible. The size of the priority queue in the merge phase and read-ahead I/Os forecast is application-dependent.

Therefore, in this work, we seek to design a scalable, high-speed specialized priority queue which can support replace operations.

III. Priority Queue Architectures

In this section we first discuss the traditional heap-based priority queue implementation in software. Then we propose three different FPGA-based priority queue architectures: register-tree, register-array and BRAM-tree. These architectures take advantage of the parallelism of the FPGAs to allow swap (compare-and-swap) operations to continue occurring at different levels of the priority queue. Their cost-performance tradeoffs are also evaluated in the end.

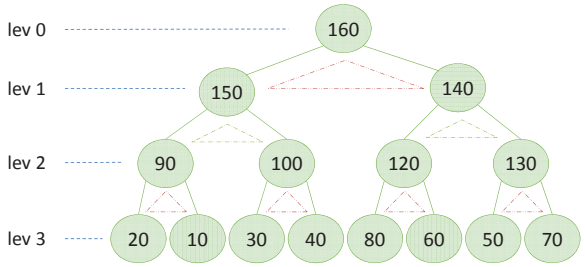

Figure 1. Register-tree.

A. Software Implementation

In a typical software implementation, the priority queue is implemented using a binary heap. An *enqueue* operation inserts the new item at the leftmost empty node, and then possibly swaps the new item with the parent node and continues upwards (heapify-up) to regain the heap property. A *dequeue* operation will get the maximum item at the root, and replace it with the rightmost non-empty node in the last level. The item that is displaced continues downwards (heapify-down), and is compared against the next level until it finds its proper place in the tree. It takes up to O(logN) time to *dequeue* or *enqueue* an item, because in the worst case an item must traverse the entire tree to move to its proper position. A *replace* is implemented as a *dequeue-enqueue* which also takes up to O(logN) time.

The following three proposed architectures, however, allow an operation following an replacement in O(1) time.

B. Register-Tree

As its name indicates, this design uses a tree of registers. Similar to a software-based priority queue implementation, it orders the nodes with respect to the heap property.

The tree is first fully loaded with entries (e.g., $2^N - 1$ entries in a N-level priority queue). For this example we will describe a max priority queue which returns the largest item in the queue; an example using a min priority queue is identical, replacing the comparison of largest with smallest.

Upon a replace operation, in one cycle the top item is removed and returned to the user. In that same cycle, all the entries at the even levels are compare-and-swapped with the entries at the odd levels (e.g. a parent node in level 2 gets swapped with its larger child in level 3 if the parent node is smaller than the child). Following that, all the entries at the odd levels are compare-and-swapped with the entries at the even levels (e.g. a parent node in level 1 gets swapped with its larger child in level 2 if the parent node is smaller than the child). The compare-and-swap operation tries to regain the heap property wherein if the parent node is not larger than both the child nodes, the parent node will swap with the larger of the child nodes.

Fig. 1 is an example of a max priority queue, where a parent node is larger than the two child nodes. Each trio

```
// 4 entry sample Priority Queue
reg reg0;
reg reg1;
reg reg2;
reg reg3;

// Top of priority queue
wire max_entry = reg0;

// New entry being replaced into the PQ
wire tmp_reg0 = get_user_entry();

// Larger of new entry, and reg1
wire max_0_1 = max(tmp_reg0, reg1);
// Smaller of new entry, and reg1
wire min_0_1 = min(tmp_reg0, reg1);
// Larger of reg2 and reg3
wire max_2_3 = max(reg2, reg3);
// Smaller of reg2 and reg3
wire min_2_3 = min(reg2, reg3);

// Larger of min(reg0,reg1) and max(reg2,reg3)
wire tmp_reg1 = max(min_0_1, max_2_3);
// Smaller of min(reg0,reg1) and max(reg2,reg3)
wire tmp_reg2 = min(min_0_1, max_2_3);

always @ (posedge clk)
begin
    reg0 <= max_0_1;
    reg1 <= tmp_reg1;
    reg2 <= tmp_reg2;
    reg3 <= min_2_3;
end
```

Figure 2. Register-array verilog pseudocode.

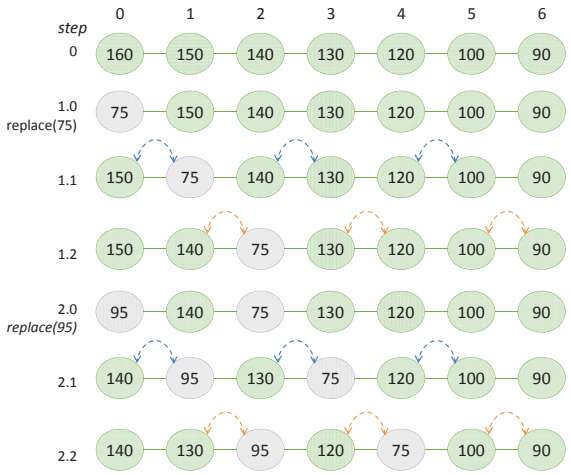

Figure 3. Register-array. The array is laid out horizontally while time progresses vertically down.

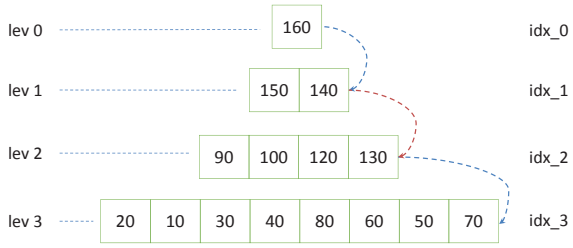

Figure 4. BRAM-tree.

denotes a compare-and-swap operation. The compare-and-swap logic in the same color operate together at the same time.

This design leverages the parallel nature of FPGAs, allowing each of these level comparisons to take place in parallel across the entire tree (e.g., level 0, 2, and 4 occur in parallel, and level 1 and 3 occur in parallel). By allowing the swaps to complete in one cycle, the register-tree can sustain a replace operation every cycle, compared to a replace operation taking O(logN) time in the software heap case.

The compare-and-swap operation can be implemented as two sequential comparisons: a comparison between the two child nodes and a comparison between the parent node with the larger of two child nodes. A practical implementation to minimize the latency of the compare-and-swap operation avoids two sequential comparisons by allocating three simultaneous comparisons: parent and left child, parent and right child, left child and right child. Another consideration is that although there is at most one item at each level that actually needs a compare-and-swap operation to verify the heap property, all trios are invoked to perform compare-and-swaps. This eliminates the indexing logic and costly (in terms of both area and latency) multiplexers which help to keep track of the newly displaced entries. Therefore, the total number of comparators is about 1.5N.

C. Register-Array

The second proposed implementation is to organize the registers in an array-like structure, which we call a register-array. The replace operation propagates to the next node in a fashion similar to the register-tree. Upon a replace operation, in one cycle, the leftmost node is replaced with a new item, then the array pulsates twice — all the even entries in the array are swapped with the odd entries, and then all the odd entries are swapped with the even entries. Verilog-like pseudocode of a 4-entry array is shown in Fig. 2, and Fig. 3 provides an illustrative example of the register-array. Note that if we adopt a "single child" policy in implementing the register tree, it is exactly the same as the register-array.

The register-array has two advantages over the register-tree: (1) the number of compare-and-swap logic is N-1, compared with 1.5N in the register-tree; and (2) register-array is more easier to place and route on FPGAs than the register-tree. As we later show in our experiments, the clock frequency of register-tree drops quickly as the priority queue size increases. A more formal explanation that can be found in [9] states that when embedding a complete binary tree into a two-dimensional plane, the maximum distance between adjacent nodes (dilation) has a lower bound of $\Omega(\frac{\sqrt{N}}{logN})$.

D. BRAM-Tree

We transform a register-tree into a BRAM-tree by packing registers at each tree level into a BRAM. In addition, each

level maintains an index indicating which item in the BRAM has just been displaced. Compare-and-swap logic first reads the index and then calculates the indexes of the two child nodes in the next level. This is illustrated in Fig. 4. The comparator logics between level 0 and level 1 calculate the child node indexes as $2 * idx\_0$ and $2 * idx\_0 + 1$.

The BRAM-tree is scalable with respect to the priority queue size in two aspects: (1) the comparator logic is on the order of O(logN), and (2) registers are packed into BRAMs, and thus we are no longer facing the problem of embedding a large binary tree into a two-dimensional plane. Note that the top levels of the tree only have a few elements, *i.e.* level 1 only has 1 element and level 2 only has two elements. Keeping these elements in registers is more area-efficient than keeping them in the fixed-sized BRAMs (18Kb or 36Kb) without affecting the scalability too much. Therefore in our implementation, the top few levels of the tree are stored in registers while the rest of tree is stored in BRAMs.

However, due to manipulations on BRAMs rather than on registers, two compare-and-swap operations now take multiple clock cycles. In our experiment, the resulting throughput is 4 clock cycles between replace operations.

Cyclic memory partitioning helps to improve the throughput to 2 cycles/replace operation since neighboring children are partitioned into different memory banks. However, our experiments show that when we adopt the memory partitioning technique, logic utilization increases by about 3x, leading to a reduced performance per area.

E. Architecture Benchmarking

Table I
PERFORMANCE AREA TRADEOFFS OF THREE PRIORITY QUEUE IMPLEMENTATIONS

	number of comparator logics	BRAM	throughput (cycles between replace op.)
register-tree	1.5N	0	1
register-array	N	0	1
BRAM-tree	logN	N	4

Here we summarize the three proposed priority queue implementations in Table. I. Throughput is measured as how often the priority queue can accept a new replace operation.

In terms of resource consumptions, the register-array and the BRAM-tree are the two most effective architectures. The register-array is more sensitive to comparator logic, while the BRAM-tree is more sensitive to the on-chip BRAM size.

We will soon show that in our experiment, the largest priority queue that we can fit on our FPGA platform uses the BRAM-tree architecture.

IV. A HYBIRD PRIORITY QUEUE ARCHITECTURE

Each of the described approaches has limitations that limit their applicability. Thus, we propose a hybrid approach that we call Hybrid Priority Queue (H-PQ). H-PQ combines the best of both register-array and BRAM-tree. It can both support a large priority queue size and can achieve a throughput close to 1 cycle between replace operations on average.

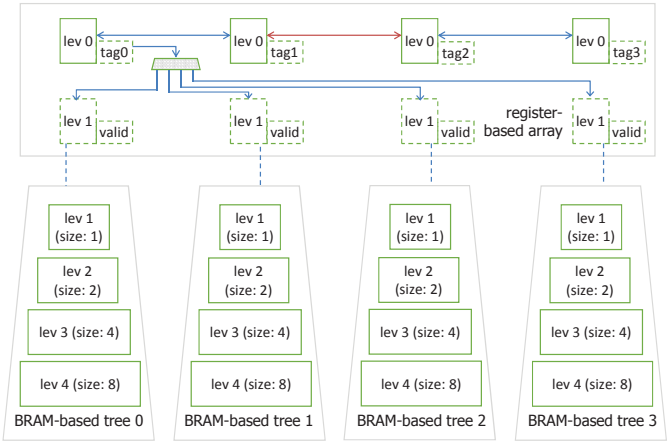

Figure 5. Hybrid priority queue (H-PQ).

A. Architecture

The design strategy is simple — we keep multiple BRAM-trees and use a register-array to sort the root nodes of the trees. The new item replaces the largest root and propagates downward in the tree to regain heap property. And then the register-array also pulsates to regain the order. The insight here is that if consecutive new items are propagating down different trees, we can then hide the slow individual throughput of the trees.

We define the max H-PQ property as follows: (1) In the register-array, a left node is larger than or equal to the right node, and (2) in the BRAM-tree, a parent node is larger than or equal to the two child nodes. When a new item replaces the leftmost node in the register-array, the H-PQ sort the root nodes of the BRAM-trees and if necessary reorder the elements in a BRAM-tree to regain the H-PQ property.

Fig. 5 is an example of four BRAM-trees with a 4-entry register-array. Each BRAM-tree has N/4 elements in an H-PQ of size N. As a side note, only the root nodes of the BRAM-trees are ordered and there is no guarantee that all elements of the left tree are larger than all elements of the right tree. The register-array is slightly modified in H-PQ to maintain the following additional information: (1) a tree tag that denotes which tree the register belongs to; (2) a buffered level 1 node of the tree (marked in dashed rectangles). The buffered node helps the comparator logic to quickly identify whether the new item needs to swap with the level 1 node without communicating with the BRAM-tree module; and (3) a valid bit of level 1 node of the tree. Once it is determined that the new item needs to swap with the level 1 node of the tree, the new item is propagated down to the BRAM-tree module and invokes the BRAM-tree module. The valid bit is marked as 0 until the new

level 1 node is determined within the BRAM-tree and sent back to the register-array.

The register-array in a max H-PQ works as follows. First, the left-most register is replaced with the new item. Based on the tree tag, the new item is locally compared with the level 1 node of the corresponding tree. If the node is marked as invalid, the comparison operation stalls until the valid bit is set to 1. In the case of a max priority queue, if the new item is larger than the level 1 node, then the new item stays at the left-most register; otherwise the new item is sent to the corresponding tree and the left-most register is updated with the level 1 node's value. After the leftmost register gets updated, the register array pulsates twice: compare-and-swap even entries with odd entries and compare-and-swap odd entries with even entries. The tree tags also swap together with the entries.

The BRAM-tree is the same as the one described in Section III-D except that level ith BRAM only contains 2^{i-1} items, rather than 2^i items. The reason for this is that we want to keep only one item in level 1, thus reducing the buffering cost in the register-array and the communication cost between the register-array and the BRAM-tree.

B. Performance Analysis

While register-array and BRAM-tree designs have data-independent throughputs (1 and 4 clock cycles between replace operations respectively), H-PQ has a data-dependent throughput – the throughput depends on how often the register-array is blocked, waiting for the newly updated level 1 node from the BRAM-tree (*i.e.* waiting for the valid bit flipping to 1). That being said, if the new item is always larger than the level 1 node, the BRAM-trees are never invoked, leading to a highest possible throughput that equals 1 clock cycle between replace operations.

To be fair, the following discussion assumes a **random** input. The size of the input data set we use is 2N, since in replacement selection sort the average number of data passing through the priority queue in each run is 2N.

Our hybrid priority queue can achieve a throughput close to 1 cycle between replace operations for two important reasons: (1) as we have mentioned, it hides the slow individual throughput of the tree when consecutive new items are dropping down different trees; and (2) the compare-and-swap between level 0 nodes and level 1 nodes are now separated into two stages: compare and, only if necessary, swap. Only swap will invoke the BRAM-trees. As more items pass through a max (min) priority queue, the items that remain in the priority queue tend to be smaller (larger). Then the new item is more likely to be larger (smaller) than the existing items in the BRAM-trees thus would only affect the entries in the register-array. The BRAM-trees are invoked less frequently, thereby being less likely to stall the whole system.

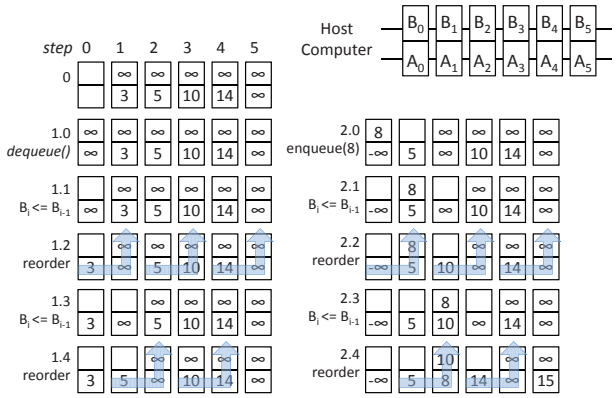

Figure 6. Systolic array.

V. Related Work

A. Pipelined Heap

There are multiple pipelined heap (priority queue) implementations in the literature; [4] and [5] are the most similar to our BRAM-tree implementation. Their design is targeted at network routers and they focus on *enqueue* and *dequeue* operations rather than the *replace* operation. [4] can sustain a replace operation every 6 clock cycles. [5] can sustain a replace operation every 2 clock cycles by using two 4-entry-wide port memories (256-bit, given 64-bit entries) and local bypass logic. [5] also argues that achieving a throughput of 1 clock cycle between replace operations requires global bypass paths — each level of the heap must have bypass paths from all the levels below it; the number of bypass paths is O(logN * logN), which is considered too expensive.

Our BRAM-tree implementation can sustain a replace operation every 4 clock cycles but has a simple design that only requires single-entry-wide BRAMs. Moreover, our hybrid priority queue implementation can sustain a throughput that is close to 1 clock cycle between replace operations on average.

B. Systolic Array

The other implementation is a systolic array [6], which pairs two arrays, A and B, together. Fig. 6 is an example of an 5-entry array. The arrays are of equal size, N+1, resulting in a total space requirement of 2(N+1). Items are kept in *sorted order* in the primary array A, and when items are inserted, they are placed in the B array. Items are compared across the B and A arrays to identify whether items must be shifted in the sorted array A to regain ordering, or transferred to the B array. Across clock cycles array B is shifted by one element.

Fig. 6 illustrates a systolic array for a min priority queue. The even and odd numbered processors pulsate alternately, each time executing the following: (1) $B_i \leftarrow B_{i-1}$; (2) arrangement of the elements in A_{i-1}, A_i, and B_i so that

$A_{i-1} \leq A_i \leq B_i$. The array pulses twice each time. To enqueue, the new item and $-\infty$ are placed in B_0 and A_0 respectively. To dequeue, $+\infty$ is placed in both B_0 and A_0.

From the host computer's standpoint, *enqueue* and *dequeue* are constant-time operations. The main drawback is that the systolic array needs two arrays—twice as much storage as our register-array. Also, the systolic array takes a total of six comparisons at each stage, while our register-array only needs two comparisons.

C. Shift Register

The shift register-based implementation is similar to the insertion sorting algorithm. It keeps the current items in a sorted array of registers, with control logic (a processing engine) attached to each register. Upon *enqueue*, a new item is broadcast to all the control logic. Each control logic compares the new item with its register and then makes a local decision about where to insert the new item. All the registers that are smaller than the new item shift one step backward. Upon *dequeue*, all the registers shift one step forward.

The shift register has the benefit of requiring only one array of registers. However it suffers from a bus loading problem during the broadcast stage, as the new item needs to be routed to the input of each control logic. Thus, the shift register not a scalable design and is only suited for small problems.

D. Hybrid Shift/Systolic

A hybrid shift register and systolic array based design is presented in [10], where the shift register is considered to be the basic building block of the systolic array design. Instead of a single item, each systolic block holds c number of items working together as a shift register design.

The storage size of hybrid shift/systolic is $N * (1 + \frac{1}{c})$, which is less than the size of systolic array (2N). By restricting the broadcast size to c, instead of N, it is a more scalable design than the shift register.

Nevertheless, the register-array design presented in this paper requires even less storage (as small as N) and does not need any broadcast logic.

E. FIFO Priority

FIFO priority is similar to the bucket sorting algorithm, where new items are steered to the FIFO that has the same priority level. Upon *dequeue*, FIFOs are scanned in order from the highest priority level to the slowest one until a nonempty FIFO is found, and then the head of that FIFO is dequeued.

Increasing the number of priorities requires adding more FIFOs; thus the FIFO priority is only a good fit when the number of different priorities is very limited.

Figure 7. Resource consumptions. Entry size is 32-bits. Resource consumption of the 256-entry systolic array is a projected number.

VI. Experiments

In this section, we discuss the resource consumptions, performance and energy consumptions of our proposed priority queue architectures. We mainly focus on register-array, BRAM-tree and hybrid priority queue since the register-tree is dominated by the register-array on both performance and resource consumptions as we discussed in Section. III-E.

A. Experimental Setup

We use the Xilinx Zynq ZC706 FPGA board. We first design C++ templates that can adapt to different priority queue sizes and entry sizes. Then we use the Vivado_HLS (version 2013.2) to synthesize the design from C++. The Xilinx PlanAhead (version 14.3) performs the low-level synthesis. We tried different clock frequencies in PlanAhead to find the highest frequency sustained by our design. The software implementation (using STL) runs on a 2.33GHz Intel Xeon machine. The power of the FPGA and the ARM is measured from the power buses on the ZC706 board using the Texas Instrument Fusion Technology, which monitors real-time voltage and current data.

We feed random data into our priority queue. A pseudo-random number generator is implemented on FPGAs since it is too slow to have ARM to generate the random numbers and send the numbers to FPGAs over the AXI bus. The pesudo-random number generator we use is "linear feedback shift register". The correctness of our design is verified by ARM when the input and output of the priority queue are both redirected to the ARM.

B. Register-Array and Systolic Array

In Fig. 7 we show the resource consumption of our register-array and the systolic array under different configurations. In both cases, BRAM consumptions are 0% and the designs are LUT-bounded. The systolic array has a much higher resource consumption than the register-array. When entry size is 256, the systolic array does not fit on chip.

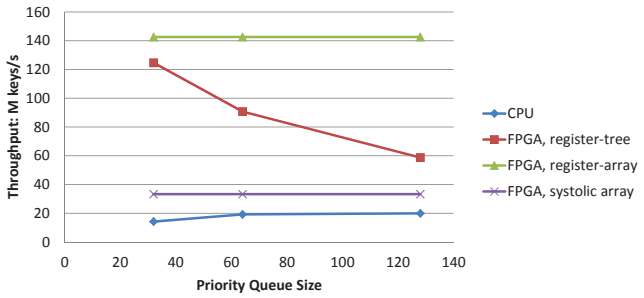

Figure 8. Throughput of priority queue. Entry size is 32-bits.

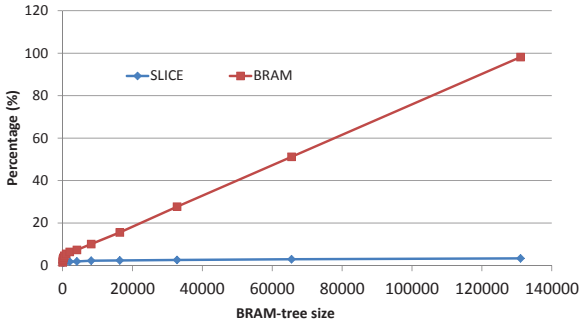

Figure 9. Resource (Slices) consumptions of BRAM-tree. Entry size is 64-bits.

Fig. 8 shows the throughput *vs.* priority queue size curve. On our FPGA platform, the largest priority queue that can fit on chip using register-array has a size of hundreds of items.

As priority queue size increases, the clock frequency of register-tree drops quickly due to its complicated structure which leads to a hard place-and-route problem. However the register-array is scalable with respect to the priority queue size and thus keeps a high performance that is about 7x faster than the software implementation. Note that the sizes of N aren't large enough to stress the algorithmic complexity in the software implementation. Thus, the throughput slightly increases as N increases. The register-array is 4.3x faster than the systolic array; this is because the systolic array has a more complex implementation of the *replace* operation.

C. BRAM-tree

The BRAM-tree is a BRAM-bounded design, as shown in Fig. 9. BRAM consumptions increase linearly with the priority queue size, while the number of slices increases very slowly as the compare-and-swap logics are on the order of O(logN).

The largest priority queue that can fit on chip using BRAM-tree has about 131,000 64-bit items.

The BRAM-tree is better suited for large priority queues, while the register-array is well suited for small priority queues.

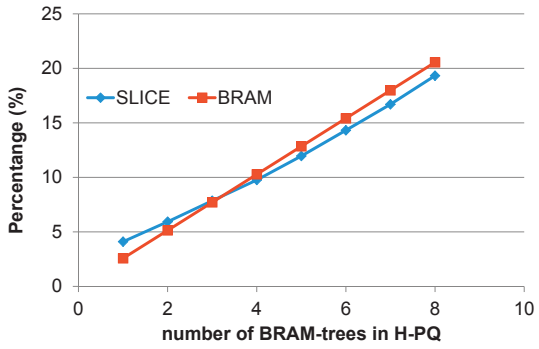

Figure 10. Resource consumptions of H-PQ. Each BRAM-tree size is 1024. Entry size is 64-bits.

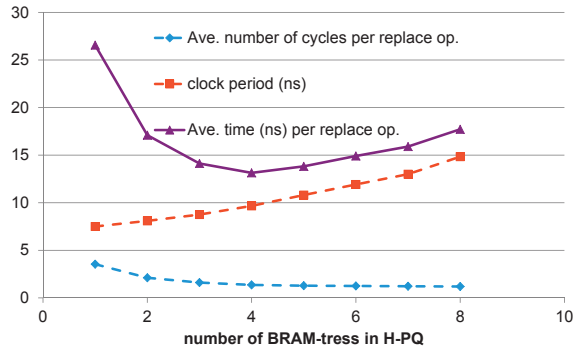

Figure 11. Design space explorations of hybrid priority queue (H-PQ).

D. Hybrid Priority Queue (H-PQ)

Fig. 10 shows the resource consumption of H-PQ when its size varies from 1024 (1 BRAM-tree) to 8192 (8 BRAM-trees).

Fig. 11 shows the design space explorations of the average throughput of our H-PQ. We use a random input data set that has 2N 64-bit items. As the number of BRAM-trees increases, the probability that the consecutive new items will fall into the same tree decreases. Thus, average throughput (number of cycles between replace operations) gets very close to 1 when the number of BRAM-trees is larger than 4. The register-array experiences a slight increase in clock period when the array gets larger. (We use the best clock frequency achieved in low-level synthesis.) This increase is due to the indexing of the tree-tag, where a multiplexer is used. Overall, the best throughput (13.1ns between replace operations) is achieved when the number of BRAM-trees is four.

Considering the resource consumptions of the BRAM-tree (Fig. 9), the projected largest H-PQ that can fit on our FPGAs has about 120,000 entries. It uses four BRAM-trees and each tree contains about 30,000 64-bit entries.

Table II
METRICS OF A SINGLE REPLACE OPERATION ON DIFFERENT PLATFORMS

	FPGA (H-PQ)	Xeon (STL)	ARM (STL)
latency (ns)	13	56	204
dynamic power (watt)	1.8	15	1.0
dynamic + static power (watt)	9.0	45	8.2
dynamic energy (nJ)	23.4	840	204
dynamic + static energy(nJ)	117	2520	1672

E. Energy Efficiency Comparisons

We compare the energy efficiency of the replace operation with the CPU implementations in Table. II. The size of priority queue is 4096 and input is an 8192 random data set. The FPGA implementation uses an H-PQ that contains four BRAM-trees. The CPU implementations that run on Xeon and the embedded ARM processor on Zynq board use the STL priority queue and are all compiled with -O3. We also implement an OpenMP multi-threaded priority queue that works in fashion similar to register-array on an 8-core Xeon node. However its throughput is much worse when comparing with the STL implementations due to the limited number of threads that can run in parallel on CPUs and overhead to set up the threads. As we can see from Table. II, H-PQ on FPGAs have the highest throughput and highest energy efficiency.

VII. ACKNOWLEDGEMENT

We would like to thank Brad Morrey, Kimberly Keeton and Harumi Kuno for their helpful inputs early on, and Janice Martin, for helping to shepherd the paper. We also thank anonymous reviewers from FCCM 2014 and FPL 2014 for their time and valuable feedback. This work is partially supported by Center for Domain-Specific Computing.

VIII. CONCLUSION

We demonstrated the design of several FPGA-based priority queues, which are specialized to support the commonly used *replace* operation. Through this specialization, our designs are able to achieve a higher throughput than traditional software-based designs, and better scalability and resource utilization than previously proposed FPGA-based designs. In particular, our Hybrid Priority Queue combines the best aspects of all our designs, and it is fast and supports a large queue. The use of high level synthesis enabled us to rapidly design and evaluate all of our priority queue implementations. Our priority queue designs can be applied to many database cases, and illustrate how FPGAs can effectively be used to improve the performance of many important systems.

REFERENCES

[1] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, p. 10, 2006.

[2] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong, "Autopilot: A platform-based esl synthesis system," in *High-Level Synthesis*. Springer, 2008.

[3] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.

[4] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2. IEEE, 2000, pp. 538–547.

[5] A. Ioannou and M. G. Katevenis, "Pipelined heap (priority queue) management for advanced scheduling in high-speed networks," *IEEE/ACM Transactions on Networking (ToN)*, vol. 15, no. 2, pp. 450–461, 2007.

[6] C. E. Leiserson, "Systolic priority queues," 1979.

[7] X. Martinez-Palau, D. Dominguez-Sal, and J. L. Larriba-Pey, "Two-way replacement selection," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 871–881, 2010.

[8] D. E. Knuth, *The art of computer programming*. Pearson Education, 2005.

[9] J. D. Ullman, *Computational aspects of VLSI*. Computer Science Press Rockville, MD, 1984, vol. 11.

[10] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *Computers, IEEE Transactions on*, vol. 49, no. 11, pp. 1215–1227, 2000.