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 both enqueue and dequeue operations.**

To improve the performance, we propose several scalable and high-speed FPGA-based implementations of a priority queue. Our insight is that the above 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. We implement three priority queue designs, including use of a register-based array, register-based tree, and BRAM-based tree, which have different benefits and trade-offs of throughput, frequency, and maximum size. More 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 21.5x energy efficiency, compared with the Xeon CPU 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 the 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.

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. 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 [2, 3] or a systolic array [4]. These approaches have several limitations:

(1) These designs use on-chip memories to store the data 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 have to be introduced so as to overlap the successive operations.

(2) These designs aim to support all three priority queue operations, *enqueue*, *dequeue* and *replace*. However, we find only the *replace* operation is needed when a priority queue is used in a database system, motivating us to design a customized priority queue.

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, a 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 highlight the practical considerations of designing priority queues for databases in Section II. Section III proposes three different priority queue architectures and provides architecture benchmarking. Section IV proposes a hybrid priority queue architecture. Experimental results and discussions are shown in Section V. And we conclude in Section VI.

Readers are referred to [5] for more extensive discussions on related work, priority queue applications in databases, proposed architectures and experimental results.

II. Priority Queue in Database Systems

In database systems, a priority queue can be used for several purposes: (1) It 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. (2) N-way merge requires proper I/O handling if the initial runs reside in external devices. We can effectively predict which sorted runs to prefetch by using

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

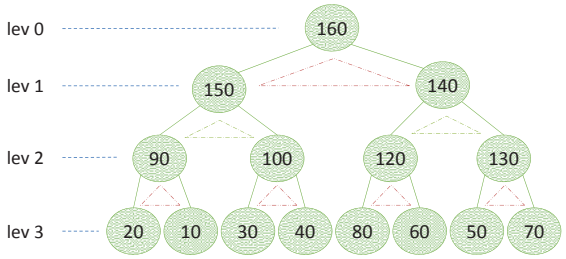

Figure 1. Register-tree.

a priority queue. (3) Replacement selection sort, also known as tournament sort, is one of the most commonly used run generation techniques for external sorting [1, 6]. The replacement selection performs the in-memory sort by passing the data through a large priority queue.

We summarize 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. 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. Their cost-performance trade-offs are also evaluated at the end of this section.

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 select 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 a 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.

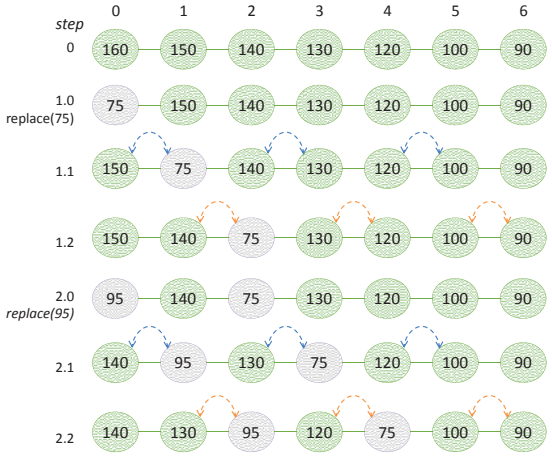

Figure 2. Register-array. The array is laid out horizontally while time progresses vertically downward.

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 the larger of its two children in level 3 if the parent node is smaller than that 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 the larger of its two children in level 2 if the parent node is smaller than that 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 denotes a compare-and-swap operation. The compare-and-swap logics 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.

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. Fig. 2 provides an illustrative example of the register-array. Note that if we adopt a "single child" policy in implementing the register tree, it becomes exactly the same as the register-array.

The register-array has two advantages over the register-tree: (1) the number of compare-and-swap logics is N-1, compared with 1.5N in the register-tree; and (2) the register-array is much easier to place and route on FPGAs than the register-tree. The clock frequency of the register-tree drops quickly as the priority queue

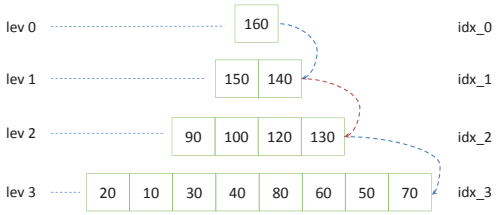

Figure 3. BRAM-tree.

size increases. A more formal explanation that can be found in [7] 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 BRAMs. 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. 3. 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.

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.

E. Architecture Comparisons

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/B	4

Here we summarize the three proposed priority queue implementations in Table. I. B is the size of a BRAM. 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.

As we experiment with our FPGA board, the largest priority queue that can fit on the FPGA uses the BRAM-tree architecture.

IV. A HYBIRD PRIORITY QUEUE ARCHITECTURE

Each of the described approaches has shortcomings 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 the register-array and BRAM-tree. It can both support a large priority queue size and can achieve an average throughput close to 1 cycle between replace operations .

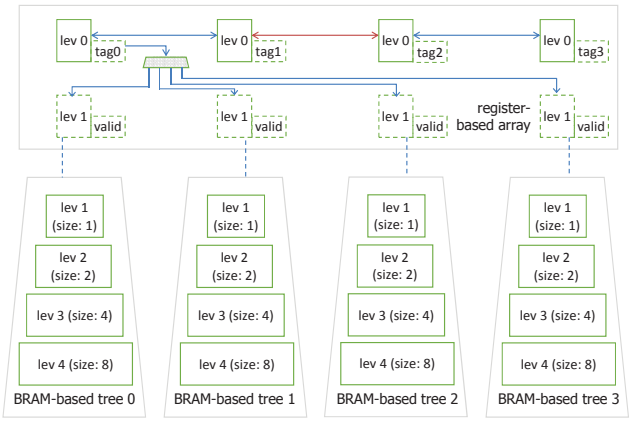

Figure 4. Hybrid priority queue (H-PQ).

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.

Fig. 4 is an example of four BRAM-trees with a 4-entry register-array. 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 the 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 compare-and-swap logics in 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 leftmost 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 leftmost 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.

B. Performance Analysis

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

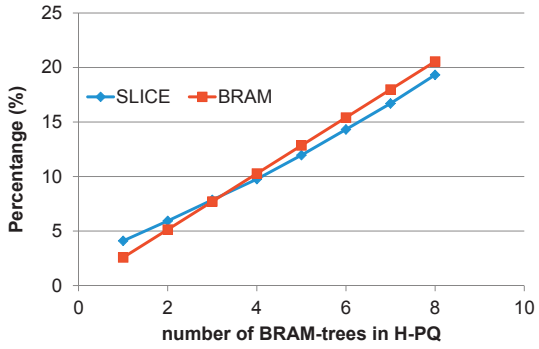

Figure 5. Resource consumptions of H-PQ. Each BRAM-tree size is 1024. Entry size is 64-bits.

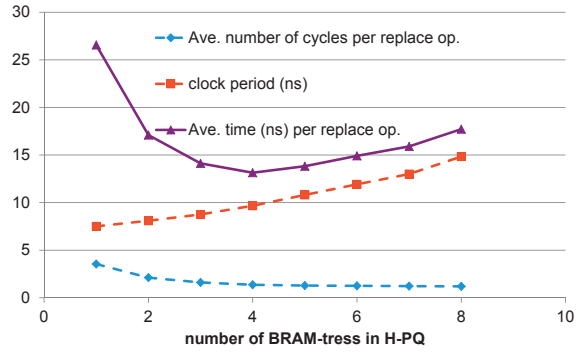

Figure 6. Design space explorations of hybrid priority queue (H-PQ).

items in the BRAM-trees, and would thus only affect the entries in the register-array. The BRAM-trees are invoked less frequently, thereby are less likely to stall the whole system.

V. Experiments

A. Experimental Setup

We use the Xilinx Zynq ZC706 FPGA board for our experiment. 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 Instruments fusion technology, which monitors real-time voltage and current data.

B. Hybrid Priority Queue (H-PQ)

Fig. 5 shows the resource consumption of H-PQ when its size varies from 1024 (1 BRAM-tree) to 8192 (8 BRAM-trees). Fig. 6 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. 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 4.

The largest H-PQ that can fit on our FPGAs has about 120,000 entries. It uses 4 BRAM-trees and each tree contains about 30,000 64-bit entries.

C. Energy Efficiency Comparisons

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

We compare the energy efficiency of the replace operation with the CPU implementations in Table. II. The size of the 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 the Zynq board use the STL priority queue and are all compiled with -O3. As we can see from Table. II, H-PQ on FPGAs has the highest throughput and highest energy efficiency.

VI. 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.

VII. Acknowledgement

We would like to thank Brad Morrey, Kimberly Keeton and Harumi Kuno for their helpful input early on, and Janice Wheeler, for helping to shepherd the paper. We also thank anonymous reviewers for valuable feedback. This work is partially supported by the Center for Domain-Specific Computing under the NSF Expedition in Computing Award CCF-0926127.

References

[1] G. Graefe, "Implementing sorting in database systems," *ACM Computing Surveys (CSUR)*, vol. 38, no. 3, p. 10, 2006.
[2] 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.
[3] 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.
[4] C. E. Leiserson, "Systolic priority queues," 1979.
[5] M. Huang, K. Lim, and J. Cong, "A scalable, high performance customized priority queue," Computer Science Department, UCLA, TR140013, Tech. Rep., 2014. [Online]. Available: http://fmdb.cs.ucla.edu/Treports/140013.pdf
[6] 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.
[7] J. D. Ullman, *Computational aspects of VLSI*. Computer Science Press Rockville, MD, 1984, vol. 11.