An efficient FPGA priority queue implementation with application to the routing problem

Joseph Rios
Technical Report ucsc-crl-07-01
Department of Computer Engineering
School of Engineering
University of California, Santa Cruz *

May 9, 2007

Abstract

The FPGA-QQ (Field Programmable Gate Array Quick Queue) is a novel, efficient priority queue implementation targeted specifically for FPGAs. This paper describes its architecture and use in acceleration of the FPGA routing problem. FPGA-QQ utilizes the FPGA's blocks of on-chip memory to store keys and values in a completely ordered fashion. The use of the on-chip block memory allows hundreds to thousands of items to be held in the queue at any given time, more than any other published design for an FPGA implementation. The queue is formed by a series of cascadable nodes. The latency of the queue is (roughly) equivalent to the depth of the RAM in a given node. This implementation is then applied to accelerate the routing phase of a standard FPGA CAD flow. Results are positive and speedups grow with the size of the queue up to (and possibly beyond) 16× faster than a standard software heap.

keywords: FPGA, routing, priority queue, hardware acceleration

*Supported by a Cota-Robles Fellowship. Thanks for lab support and guidance from Professor Pak Chan.

1 Introduction

Many applications targeted for implementation on an FPGA (Field Programmable Gate Array) SoC (System on a Chip) use the programmable fabric of the chip to accelerate portions of the application. A common data structure ripe for this sort of acceleration is the priority queue (PQ). A PQ accepts and maintains items in such a fashion that the "best" item (the one with lowest/highest cost, highest criticality, etc.) is the one returned upon a fetch from the PQ. A specific application utilizing such a PQ is the router of the open-source FPGA CAD tool VPR. Operations on the PQ (adds, fetches, resets) in VPR's router take upwards of 25-30% of the router's overall processing time. With an eye toward accelerating VPR's router, we have developed the FPGA QuickQ.

After exploring published implementations of priority queues, it became clear that those solutions targeted for ASICs (Application Specific Integrated Circuits) weren't ideal for implementation in an FPGA. The major reasons for this were the limited amount of distributed memory on the chip and the massive routing requirements. Our initial attempts to implement these designs resulted in small queues (on the order of dozens of items) that occupied the majority of the FPGA. This led to our investigation of the use of the large on-chip memory blocks present on modern FPGAs. The resulting FPGA QuickQ is parameterized, scalable and holds hundreds to thousands of items in the queue depending on the design parameters. The scalability is due to the cascadable nature of the nodes. Latency is roughly equal to the parameterizable depth of the RAM used in an individual node of the queue. Thus, adds and fetches on a queue with a node depth of 16 take 16 and 18 cycles, respectively.

The FPGA QuickQ has been completely implemented and integrated with VPR, the Versatile Place and Route tool [1], in order to measure the effects of using this PQ instead of the software heap implementation built into VPR. We also tested the FPGA QuickQ on a standalone dataset with good results. This paper describes the architecture in detail and discusses initial experimental results with VPR.

The remainder of this paper is organized as follows. In Section 2 some background is provided on hardware priority queues and the FPGA routing problem. Section 3 describes our approach to the routing problem and the tools used to in our experiments. Next, Section 4 the actual architecture of the FPGA QuickQ is described. Results are presented in Section 5 and conclusions with research in progress in Section 6.

2 Background

2.1 Hardware priority queues

There have been a good number of hardware priority queue implementations described in the literature [2, 3, 4, 5]. Bhagwan and Lin [3] designed a physical heap and organized the control as to pipeline the commands between levels of the heap. Chao and Uzun [4] describe a queue wherein new values entering the queue are broadcast to all cells and, through use of subtractors in each cell, the new value finds its

appropriate place while all displaced cells simultaneously shift. Moon, Rexford and Shin [2] solve some of the fanout problems associated with Chao's design by broadcasting only to a bundled subset of slots. The displaced value from that bundle is passed to a subsequent bundle and so on. The design is scalable, efficient and bears the closest resemblance to the QuickQ presented in this paper. The three designs discussed above were all designed for use in network ASICs. Kuacharoen, Shalan and Mooney [5] implement a PQ much like Chao's along with other elements to act as a Real-Time Operating System task scheduler. They discuss implementation in an FPGA, but have to deal with very few priority levels and a small number of items in the queue at any given time. Our QuickQ implementation is targeted specifically for FPGAs and can maintain queues numbering in the hundreds or thousands depending on parameters.

2.2 The FPGA routing problem

The PathFinder FPGA routing algorithm [6] is successful and commonly applied. It is a greedy, iterative algorithm wherein each net is rerouted each iteration. Each net takes the least cost path on each iteration and nets are allowed to share resources, i.e. two (or more) nets could be routed on, say, the same wire or use the same switch. Every time a resource is shared, the cost is increased for that resource for all future iterations. Thus, ultimately, the highly contested resources will eventually become too expensive for all but one net. The authors of PathFinder give credit to Nair for the basis of their algorithm [7].

To help bolster research in the field of FPGA CAD, an open-source program called Versatile Place and Route (VPR) was developed at the University of Toronto[1]. VPR performs placement and routing on a synthesized netslist targeted to a fully documented (and configurable) architecture. This tool has been widely accepted and used in research. The core of the routing portion of VPR is based on PathFinder. The VPR suite comes with 20 benchmark circuits for experimentation.

Hardware acceleration of routing is an active field of research and has been so for some time[8]. Of the more recent publications, several have come from the California Institue of Technology and UC Berkeley [9, 10, 11], often in partnership. The approach to accelerating routing by these researchers involves a completely novel implementation of the physical routing resources [10, 11] on the FPGA. Their premise is to have the routing architecture designed to support a physical implementation of the PathFinder algorithm [6]. DeHon, Huang and Wawrzynek propose an FPGA with routing resources that allow the search for routes between placed blocks to be done in the hardware itself, a so-called "spatial router". Thus, a design that has already been run through a placer is physically placed onto the FPGA and then allowed to find a routing for itself. Special switchboxes are the novel idea of this research. These switchboxes contain extra logic to account for congestion approximation. This system doesn't run a strict version of Pathfinder, but their approximation of the algorithm in hardware reportedly gives results within 3% of the software version. The reported speed-up varies from 211× to 133,131×.

The approach to routing acceleration described here differs from all others mentioned above. We propose to use standard FPGAs, likely the target FPGA itself, in the CAD process.

3 Approach

Starting with VPR 4.3, we compiled the code with -pg and -g flags. Then each of the 20 benchmark circuits were placed. The placement files were used to run VPR again, this time with the `-route_only` flag and gprof was used to analyze the results. All of this was accomplished on a standard workstation PC running Linux. The major heap operations, `get_heap_head() and add_to_heap()`, accounted for about 25% of the total runtime. Figure 1 summerizes this information. After the combined heap operations, the next largest chunk of time was spent in the `timing_driven_expand_neighbours()` function. The percentage of time spent in this function was also in the 20-30% range. Given this information about VPR's runtime, the decision was made to improve the speed of the heap operations in hardware.

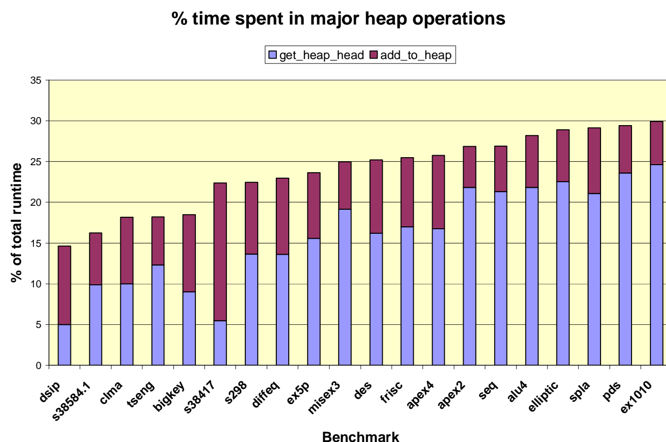

Figure 1: **Percentage of time spent on the major heap operations obtained via gprof.**

3.1 Tools

We ported VPR version 4.3 to the Xilinx ML310 Development Board [12] which features a VirtexII Pro FPGA and 256MB of DDR RAM. This FPGA has two hard PPC405 cores. For the purposes of this research, one of the PPCs was used as the CPU. Xilinx' filesystem (xilmfs) and mfsgen tool were used to create the necessary files for VPR to read and a location to write files resulting from the various runs. These files included the netlist and the placement file for that netlist. The filesystem was downloaded to the external RAM along with most of VPR. Some of the code (mostly for interrupts and boot code) was located on chip. Most of the changes to VPR involved interactions with this file system. We ran VPR in a standalone environment on the board (no operating system). We received output (and provided occasional input) via an RS232 connected UART.

4 FPGA QuickQ

The FPGA QuickQ works by inserting items into a sorted list by comparing the new item with each item already present starting from the head of the queue. The queue is divided into nodes of equal size. Each node has its own dual-ported RAM of the appropriate depth and width to store keys and, potentially, some associated data. The on-chip RAM needs two ports to support a simultaneous read and write. Except for the first two cycles of an add to the queue, there is a read and/or a write on every clock cycle. Reads and writes always occur in different locations, so there is no concern of the ordering of the reads and writes (i.e. write-before-read, etc.). When the new item finds its appropriate location in the queue, it displaces the current resident of that location. The displaced item is placed in the next location and this displacement continues until the end of the locations in the current node. The last displaced item (or the new item if it doesn't find its place) is passed to the next node along with some control information. In our current design, the last node will simply lose the last item of a full queue. Figure 2 shows this process on a node with depth 4. The new key enters the node and comparisons and displacements continue until the last value is passed to the next node. Note that in figure 2 the key *is* the data. It is possible to attach another value or values with the key, then this entire string of bits representing the key and data would be passed along the queue, though comparisons are made only on the key. Likewise, a secondary key could be included. These options need only be implemented in the Value Router seen in figure 3. The Value Router determines which key in the current comparison is written into the current location in the RAM and which moves on to be compared with the next key. The key that moves on to be compared with the next location is temporarily stored in a register (Temp Register in figure 3) until the next clock cycle/read from the RAM.

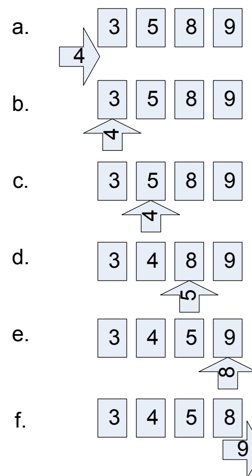

Figure 2: **An insertion into a sorted node.** Note the exiting '9' is passed onto the next node for insertion.

The default values of the queue are determined by the comparison chosen. For example, on a min queue (where the smallest key is always at the head), the default value would necessarily by some maximal value (say, all 1's if the keys are unsigned integers).

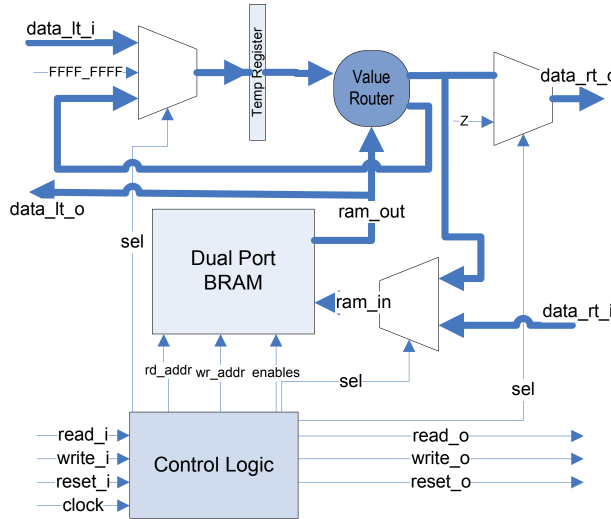

Figure 3: **A QuickQ node.**

5 Experiments and Results

To verify functionality and performance, we initially set the width of the QuickQ to 32 (width of an unsigned int), set the node depth to 16 and the number of nodes to 40 for a total queue size of 640 (16×40). With these parameters 1 of the 136 on-chip block RAMs was consumed per node. The QuickQ was attached to the PLB bus in our SoC (see figure 5). The PPC processor and PLB bus speeds were kept equal at 100MHz. A software heap nearly identical to the one provided with VPR was implemented. A list of 600 pseudo-random unsigned integers was generated. A sublist of these numbers was fed to each the software heap and the hardware QuickQ in turn and then read back. Using the PIT (Programmable Interval Timer) on the PPC, we were able to get cycle-accurate readings on the time to complete functions or any series of instructions. The ratio of cycles to complete the software operations versus the hardware operations increased as the dataset size increased. Figure 6 summarizes the results. The divergence in performance between the hardware and software versions was expected. As the software heap grows, it must perform more and more comparisons on each add or fetch call due to the increased depth of the heap. The QuickQ works at constant speed on additions and fetches and is, thus, unencumbered by a larger queue size. In fact, the ratio of clock cycles to nodes added/subtracted remains constant ($\approx$46) for the QuickQ, but increases steadily for the software heap (from $\approx$137 to $\approx$212).

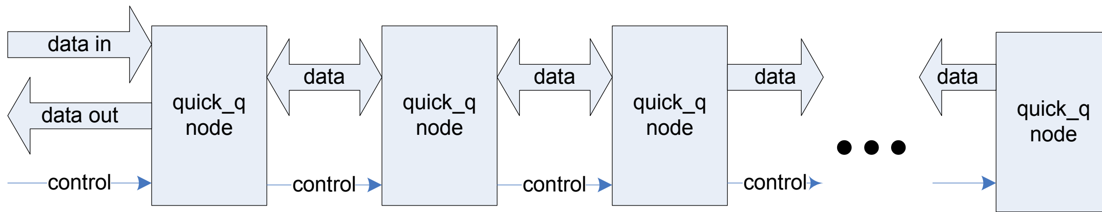

Figure 4: **A complete FPGA QuickQ consisting of any number of cascaded nodes.**

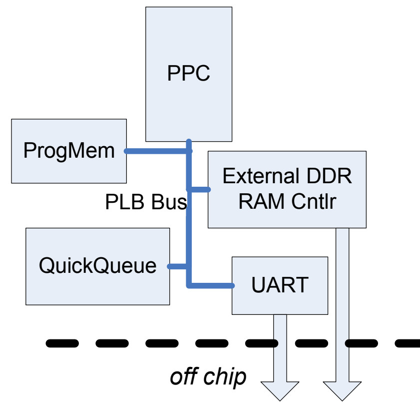

Figure 5: **Our testbench system on the ML310 board.**

The larger test came on incorporation of the QuickQ with the VPR SoC. The QuickQ here was configured with a node depth of 16 as 16 to 18 cycles seems to be the right amount of cushion between reads and writes and makes for relatively efficeint usage of the FPGA resources (logic and block RAM). Further experimentation will be required to verify these heuristics. For VPR, the key is not the same as the value that is to be returned upon a fetch. VPR sorts the values on the heap based on a float cost value, but expects a pointer to the item with the lowest cost as there is other data associated with that cost that is important for further processing. Thus, the QuickQ must maintain the 32 bit float value along with a 32 bit pointer that is to be returned on a fetch. Therefore, the QuickQ was parameterized with a width of 64 bits for this experiment. Now each node consumes 2 of the 136 block RAMs, so given the program memory requirements on-chip, we had the option to use 47 nodes, but implemented 45. This gave a maximum queue size of 720 items.

For experementation a design of size 12×12 was routed. While VPR ran successfully, it was difficult to compare the results with the standard embedded version of VPR due to different routings obtained. The differences were due to the way in which the two systems maintain their PQs. QuickQ gives priority to more recent items placed in the queue when there is a tie in cost. VPR cannot maintain this type of structure without adding complexity and/or data to the software heap, thus the QuickQ and the software heap will

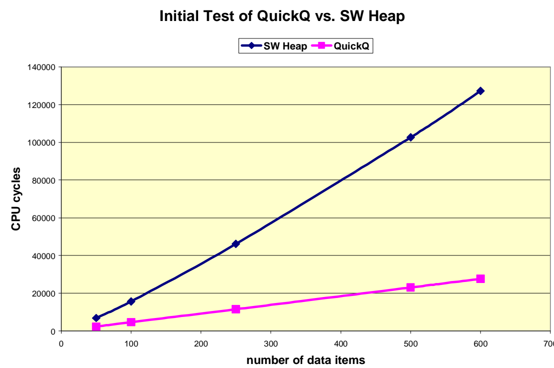

Figure 6: **A simple, initial performance test of the QuickQ versus a standard software heap.**

(almost) always discover different routings. The routings that they each discover are always comparable. The routings often had the same number of tracks, and only differed by at most one. These different search paths make it impossible to gather meaningful information from the runs as far as cycle-by-cycle or aggregate execution is concerned.

To work around this problem and gather some meaningful results, a group of artificial heap items was generated with random costs (the sorting key). These heap items were then added to each respective PQ using the same function call that VPR uses to maintain its heap and that the QuickQ would use to maintain its PQ. The number of items added to the queue was chosen to provide the software heap with a full heap on each run, thus the experiments were run with 1, 3, 7, 15, 31, 63, 127, 255 and 511 items (every heap depth of 0 through 8). After all of the additions, the same number of removals were made. The number of cycles elapsed was then calculated. After each of these runs, the validity of the returned data was verified. Since the QuickQ had a maximum capacity of 720, the largest "full heap" that could be placed in the QuickQ and reliably retrieved was 511. However, since the data returned from the PQ was not going to be used, the QuickQ could be allowed to return bad values (after the 720th value is fetched) without loss of generality to the performance of the QuickQ. Thus, the experiment was continued up to a heap depth of 16 (131071 items). Results for the verifiably correct runs (up through depth 8) are presented in figure 7 and the extrapolated results (the unverifiable runs of depth 8 through 16) are presented in 8. Note that the rate of growth in runtime continues at the same rate in both data sets. The QuickQ at depth 16 is over 18× faster than that of the software heap. It is around this depth that many of the benchmark circuits need to reach to find a solution to the routing problem. Some of the larger circuits may reach a depth of 18 on some iterations.

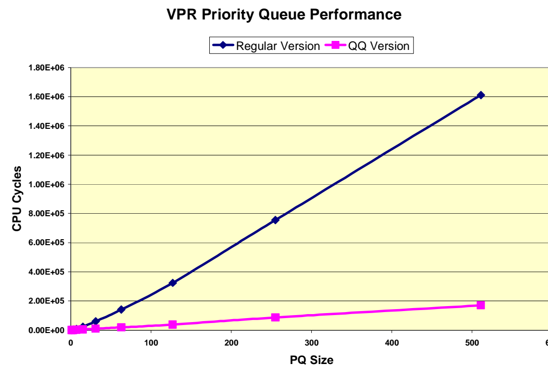

Figure 7: **Results on the VPR platform for verifiable PQ operations.**

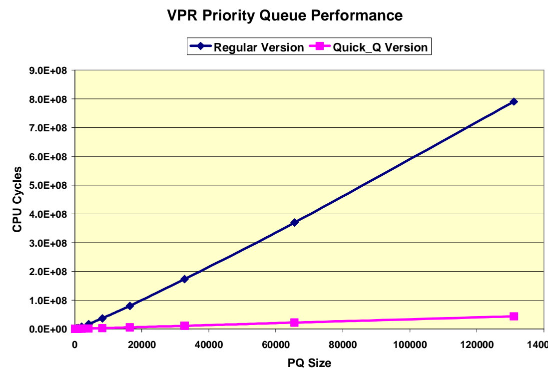

Figure 8: **Results on the VPR platform for extrapolated PQ operations.**

6 Conclusion

With further design, it will be possible to reduce this width by about half. The 4GB of address space afforded by the 32 bit pointer is excessive. Since this is an embedded system, we have control over all aspects of its operation. If all the pointers that end up in the queue are forced to point to a restricted address space, a shorter value could be stored in the QuickQ and appended with a known prefix upon its return from the QuickQ. Likewise, the 32 bit float may not be necessary either. With careful examination, that value could be quantized to 16 bits (or less?) without loss of information. By reducing the width in this manner, we would be afforded extra length in the QuickQ. Essentially, if the width is halved, the depth could be doubled (depending on the organization of the block RAM).

Also with further design, it will be possible to more exactly characterize the QuickQ's impact on VPR. Specifically, removal of many, if not all, of the print statements until after routing is complete will allow a better measure of the actual speedup of the system when using the QuickQ.

A final design improvement will be to incorporate an interface between the FPGA QuickQ and some external memory. This will allow effectively larger PQs. It is unlikely that items will be kept completely sorted in off chip memory, rather a "bucket" system could be implemented wherein closely related values could be grouped. Then with some coordination with the QuickQ, the external RAM could be triggered to perform some insertions into the on-chip queue without any processor intervention.

Despite the majority of the block RAM being dedicated to the QuickQ, there is significant logic available to implement further hardware acceleration of other CPU-intensive functions in routing. As mentioned earlier, a good candidate for such acceleration might be the `timing_driven_expand_neighbours()` function. In our lab, we have also begun to investigate the FPGA-based acceleration of the VPR placement algorithm. That research looks promising.

Overall, it is hoped that this direction of study will ultimately allow for FPGAs to be incorporated into their own CAD flow. Engineers working with FPGAs will often need to sit and wait for a significant (up to hours and possibly days) for large designs to pass through the synthesis, placement and routing stages. All the while there is an FPGA (or two or three) sitting idle on the desk or bench. If FPGA CAD and design tools could be developed with a mind toward utilizing all the resources likely available (i.e. FPGAs themselves), design turnaround time may be significantly lessened.

In this paper an efficeint implementation of a priority queue targeted specifically for FPGAs was presented. The queue was then applied to the FPGA routing problem with encouraging results. The concept of standard FPGAs being used "in-the-loop" of the standard FPGA CAD flow was also initiated.

References

[1] V. Betz and J. Rose, "VPR: A new packing, placement and routing tool for FPGA research," in *Field-Programmable Logic and Applications*, W. Luk, P. Y. Cheung, and M. Glesner, Eds. Springer-Verlag, Berlin, Sept. 1997, pp. 213–222.

[2] S.-W. Moon, J. Rexford, and K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEETC: IEEE Transactions on Computers*, vol. 49, 2000.

[3] R. Bhagwan and B. Lin, "Fast and scalable priority queue architecture for high-speed network switches," in *INFOCOM*, 2000, pp. 538–547.

[4] H. J. Chao and N. Uzun, "A VLSI sequencer chip for ATM traffic shaper and queue manager," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1634–1642, November 1992.

[5] P. Kuacharoen, M. Shalan, and V. J. Mooney, "A configurable hardware scheduler for real-time systems," in *Engineering of Reconfigurable Systems and Algorithms*, T. P. Plaks, Ed. CSREA Press, 2003, pp. 95–101.

[6] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *FPGA*, 1995, pp. 111–117.

[7] R. Nair, "A simple yet effective technique for global wiring," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, no. 6, pp. 165–172, March 1987.

[8] T. Blank, "A survey of hardware accelerators used in computer aided design," *IEEE Design Test of Computers*, vol. 1, no. 3, Aug. 1984.

[9] M. G. Wrighton and A. DeHon, "Hardware-assisted simulated annealing with application for fast FPGA placement," in *FPGA*. New York: ACM Press, 2003, pp. 33–42.

[10] A. DeHon, R. Huang, and J. Wawrzynek, "Hardware-assisted fast routing," in *FCCM*, 2002, p. 205.

[11] R. Huang, J. Wawrzynek, and A. DeHon, "Stochastic, spatial routing for hypergraphs, trees, and meshes," in *FPGA*. New York: ACM Press, Feb. 23–25 2003, pp. 78–90.

[12] Xilinx, http://www.xilinx.com/ml310.