Accelerating Graph Algorithms with Priority Queue Processor

Ch'ng Heng Sun[1], **Chew Eik Wee**[2], **Nasir Shaikh-Husin**[3], **Mohamed Khalil Hani**[4]

VLSI-ECAD Research Laboratory (P04-Level 1),
Microelectronic and Computer Engineering Department (MiCE),
Faculty of Electrical Engineering (FKE),
Universiti Teknologi Malaysia.

[1]chnghengsun@yahoo.com.sg
[2]eikweechew@hotmail.com
[3]nasirsh@utm.my
[4]khalil@fke.utm.my

Abstract

*Graphs are a pervasive data structure in computer science, and algorithms working with them are fundamental to the field. Of the various graph algorithms, techniques for searching a graph are the heart of graph algorithms. Many graph algorithms are organized as simple elaborations of basic graph searching algorithms. For the searching of a graph, Priority Queue is used to maintain the tentative search result and choice of priority queue implementation would significantly affect the run-time and memory consumption of a graph algorithm. In this work, we demonstrate how to accelerate graph algorithms using priority queue processor. Dijkstra's algorithm is chosen as the target implementation, as many state-of-the-art graph algorithms use Dijkstra's algorithm at the heart of their computational engine. Assuming embedded hardware-software co-design, results show that our priority queue processor performs better than software implementation, and the run-time complexity of Dijkstra's algorithm is reduced from O(**n** lg **n**) in software implementation to O(**n**) with our priority queue processor.*

Keywords:

Graph Algorithms, Shortest Path Routing, Priority Queue, Priority Queue Processor

1. Introduction

Graphs are a pervasive data structure in computer science, and algorithms working with them are fundamental to the field, among others; Depth-First Search, Breadth-First Search, Topological Search, Spanning Tree Algorithms, Dijkstra's Algorithms, Bellman-Ford Algorithms, Floyd-Warshall Algorithm, etc. In real world applications, there exist many algorithms which are actually extended from these basic graph algorithms. For instance, in the field of VLSI physical design automation, there are many interesting computational problems defined in terms of graph; among others, Lee's Algorithm, Maze Routing Algorithms, Matching Algorithms, Min-Cut and Max-Cut Algorithms, Minimum Steiner Tree Algorithms, Span Minimum Tree Algorithms, Clock Skew Scheduling, Clock Net Synthesis, Critical Net Routing, etc.

A graph, G = (V, E) consists of |V| numbers of vertices/nodes and |E| numbers of edges. Real world problems modeled in mathematical ***set*** can be represented as graphs, where elements in the set are represented by vertices, and the relation between any two elements are represented by edges. The run-time complexity and memory-consumption of graph algorithms are expressed in terms of |V| and |E|. A graph searching algorithm can discover much about the structure of a graph and many graph algorithms are organized as simple elaborations of basic graph searching algorithms [1]. Searching a graph means systematically following the edges of the graph so as to visit the vertices of graph. ***For the searching of a graph, Priority Queue is used to maintain the tentative search result and choice of priority queue implementation would significantly affect the run-time and memory consumption of a graph algorithm*** [2].

Priority Queue is an abstract data structure to maintain a set of elements, where all elements are arranged in accordance to their associate-priority. The associate-priority can be given as time-of-occurrence, level-of-importance, physical-parameters, delay/latency, etc, depending on target application. Two basic operations are supported by priority queue, namely (i) **INSERT(*Q, x*)**, which is generally referred to as **ENQUEUE** operation, and (ii) **EXTRACT(*Q*)** which is sometimes referred to as **DEQUEUE** operation. The performance of priority queue operations are measured in terms of **n**, where n refers to the total number of elements in the queue.

In many advanced algorithms where items/tasks are processed according to a particular order, priority queue has proven to be very useful. For ***task-scheduling*** on a multi-thread, shared-memory computer; priority queue is used to schedule and keep track of the prioritized pending processor tasks/threads. In the case of ***discrete-event-simulation*** (DES), priority queue is used where items in the queue are pending-event-sets, each with associated time-of-occurrence that serves as priority; many simulators, emulators, and synthesizers are designed based on DES concept. In the case of shortest-path based graph problem, priority queue has been used extensively in ***QoS internet packet routing***, ***weighted shortest path problem***, ***multi-constrained routing***, ***VLSI routing***, ***PCB routing***, etc.

In this paper, choices of priority queue implementation will be discussed in section 2. Section 3 introduces the priority queue processor in this work to accelerate the graph algorithm. Section 4 illustrates the Dijkstra's Algorithm. The embedded system architecture featuring the priority queue processor is described in section 5. Section 6 presents the result of this work.

2. Choices of Priority Queue

As shown in Figure 1, operation **INSERT(*Q*, *x*)** inserts a new element, ***x*** into priority queue, ***Q***; meanwhile **EXTRACT(*Q*)** returns the element with the highest priority. Preferably, in some circumstances where the highest priority is given by a minimum value, the term **EXTRACT-MIN(*Q*)** is used instead of the more general term **EXTRACT(*Q*)**; whereas in other cases where the highest priority is the maximum value, the term **EXTRACT-MAX(*Q*)** is used. Hereinafter, **EXTRACT(*Q*)** is used interchangeably with **EXTRACT-MIN(*Q*)** or **EXTRACT-MAX(*Q*)**.

INSERT (*Q*, *x*)
➔ Insert ***new element, x*** into queue Q.

EXTRACT (*Q*)
➔ Remove and return ***the highest priority element*** in queue Q.

Figure 1. Basic Priority Queue Operations

For works done to implement a priority queue, either one of **EXTRACT-MIN(*Q*)** or **EXTRACT-MAX(*Q*)** function will be implemented, depending on what application is targeted. While software implementation could be easily modified to switch from **EXTRACT-MIN(*Q*)** to **EXTRACT-MAX(*Q*)**, or the other way round, it is not the case for full-custom hardware implementation. Anyway, this is not a big issue since maximum can be treated as reciprocal to the minimum, or vice-versa (maximum = 1/minimum). For instance, given a priority queue which only provides **INSERT(*Q*)** and **EXTRACT-MIN(*Q*)**, but the target-application needs **EXTRACT-MAX(*Q*)**, then simply solve it by inverting all the associate-priority (1/priority) before insertion into queue.

The simplest way to implement a priority is to keep an associate array mapping each priority to a list of items with that priority, see Figure 2. The priorities are held in static array which stores the pointers to the list of items assigned with that priority. Such implementation is static, for example, if the allowed priority ranged from 1 to 4,294,967,295 (32-bit) then an array of (4 Giga-length) * (size of pointer storage, i.e. 32-bit) is consumed, a total of 16 Gigabytes is gone, just to construct a priority data structure.

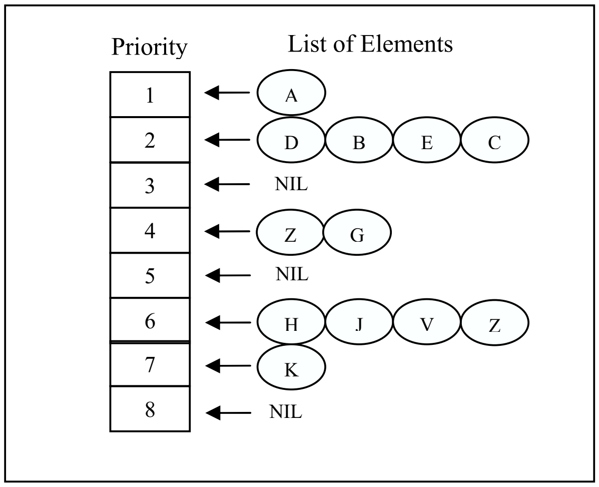

Figure 2. Simplest way to implement Priority Queue

A more flexible and practical way to implement a priority queue is to use dynamic array/queue, see Figure 3(a), which means the length of array does not depend on the range of priority. Each **INSERT(*Q*, *x*)** will extend the existing queue-length by one unit (n ← n + 1), put in the item with priority ***x***, then sort the whole queue to ensure the highest priority item is ready if **EXTRACT(*Q*)** is invoked. Such sorting during insertion will take up to O(***n***) computation run time. For extraction, since the whole queue already sorted during insertion, extraction takes constant O(***1***) time.

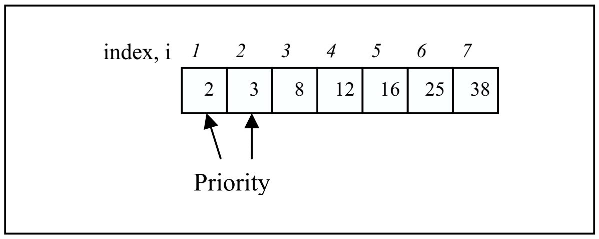

Figure 3(a). Priority Queue, implemented as array.

If a self-balanced Binary-Heap is used, see Figure 3(b), each **INSERT(*Q*, *x*)** or **EXTRACT(*Q*)** take O(**lg *n***) run-time. In order to achieve better priority queue performance, many novel approaches have been taken to implement different structure of heap, i.e. Binomial-Heap, Fibonacci-Heap, Relaxed-Heap, Parallel-Heap, to name a few. A developer has a number of choices with different access speed, memory consumption, and required hardware platform. Theoretically, Fibonacci-Heap Priority Queue ("FHPQ") has better performance compared to other types of heap, especially for applications where the number of EXTRACT and DELETE operations are relatively small compared to INSERT and DECREASE-KEY operations; however, the drawback of Fibonacci-Heap is its complexity in practice with larger constant-time-factor.

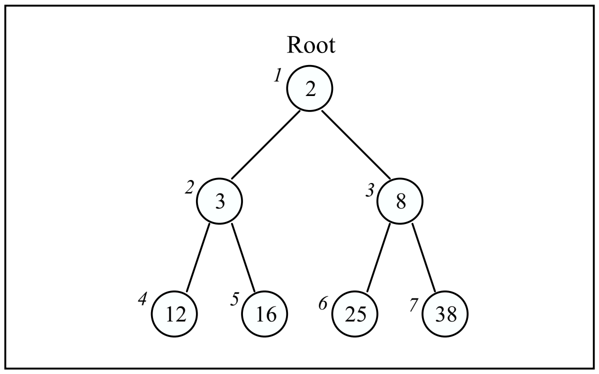

Figure 3(b). Priority Queue, implemented as heap. (in this case, Binary-Heap)

Due to the tremendous application of priority queue, there are a lot of researches done on how to accelerate the priority queue operation using custom hardware resources. There are two main categories of this work; (a) proposed full-custom parallel hardware design to accelerate array-based priority queue, (b) no specific hardware designed, but assuming PRAM (Parallel Random Access Machine) model, and propose new heap data structure executing on these PRAM. The former often achieves very high throughput and clocking frequency, favorite choice to high speed applications such as QoS network routing and real-time applications; whereas the latter often ignore the severe memory communication overhead and gives theoretical improvement in terms of run-time complexity.

Due to limited space, works in (b) is less related to our design work, it will be excluded, reader could refer to [3] for details. On the other hand, works in (a) includes ***Binary Trees of Comparator*** by [4]; such hardware priority queue is not efficient if the queue size is large, large fan-in for Binary-Tree also caused severe bus loading effect when it comes to physical implementation. Another approach to implement priority queue is using ***FIFO Priority Queue*** as proposed by [5] and [6], but the main drawback is the priority range is not flexible, just like the illustration in figure 2, thus not practical if the allowed priority range is large. ***Shift Register*** implementation of priority queue [6], [7] has better performance compared to Binary Trees of Comparators and FIFO Priority because the priority level and the queue size can be easily scaled, but Shift Register also suffers severe bus loading effect when it comes to real hardware implementation. To overcome the bus-loading effect of Shift Register, A hybrid design of ***Systolic-Shift-Register*** priority queue processor had been proposed by [8], which claims to reduce bus loading effect and could have achieved better clocking rate. Having said that, all these designs are implemented on hardware high-speed network routers, they are different from embedded platform requirement; such as in our context, graph algorithm computation (or any other DES). For example, all these designs are targeting small number of priorities (i.e. 8-bit), but in the case for graph algorithm; the data which represents the priority can be in any range (i.e. 32-bit); furthermore, for embedded system implementation, those device are practically not accessible to us.

3. Systolic Array Priority Queue Processor

In a programming model, priority queue can be viewed as an array; when new element is inserted, it will be compared with all other elements in queue, in order to determine where to slip in this new element, in such a way that priority-order of the queue is always maintained. This approach is known as insertion-sort priority queue. Derived from the insertion-sort algorithm, O(***n***) run-time per operation is needed for a priority queue with ***n*** elements.

Given an all-pair-single-source shortest-path graph problem to be solved with the Dijkstra's algorithm, the run-time complexity is O($\boldsymbol{n^2}$) if insertion-sort priority queue (hereinafter called "ISPQ") is used; the run-time can be improved to O(***n lg n***) if the priority queue is implemented as binary-heap priority queue ("BHPQ"). This shows that different choices of priority queue implementation could significantly affect the overall performance of graph algorithm, especially when the queue size is very large.

An in-house-developed hardware, Systolic Array Priority Queue Processor ("SAPQ") could further reduce the run-time complexity of the said graph algorithm to O(***n***). The SAPQ, benefited from parallelizing a modified version of insertion sort algorithm supports both **INSERT(*Q, x*)** and **EXTRACT(*Q*)** in constant O(***1***) run time. Improved from the Systolic-Shift-Register by [8], the architecture of SAPQ is much simpler, localizing control-unit and data-path for each processing element (PE) making it highly pipelined, easy for cascading, and with no bus-loading effect.

The SAPQ consists of an array of ***identical*** processing elements (PEs), with each PE holding a single queue-element. The identical-PE feature of SAPQ makes it very flexible, where a ***parameterized*** design always allows generation of any queue-size in a FPGA environment. A very large queue-size can be achieved by cascading multiple FPGAs on a single board or distributed through multiple boards. The SAPQ employs **n** number of PEs for worst-case **n**-size priority queue. Each PE is interconnected to only its immediate-neighbours. Figure 4 illustrates the top-level architecture of SAPQ. Of all the PEs, only the left-most PE will communicate with the outside world, meaning the new element is inserted to PE1, as well as the highest priority element will be extracted from PE1.

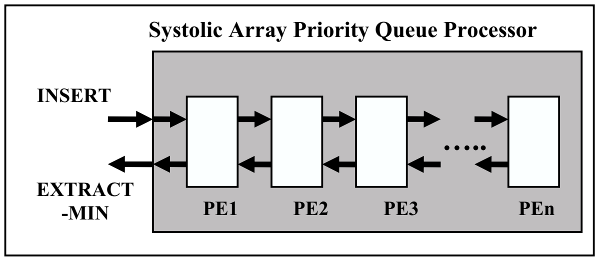

Figure 4. Systolic Array Priority Queue Processor

Each INSERT and EXTRACT-MIN operation on SAPQ completes in 3 clock cycles. The design of SAPQ allows, for each queue element, the element-ID in 32-bit, so as the associate-priority in 32-bit. The element-ID can be treated as associate-pointer which points to a block of satellite-data of that element.

The SAPQ design is compiled and targeted to ALTERA Stratix and Cyclone II FPGA devices. The synthesis shows that the design can obtain 240 MHz clocking frequency in Stratix and 175 MHz in Cyclone II. The design achieved 5.12 Gbps throughput rate for Stratix and 3.73 Gbps in Cyclone II. For details of SAPQ design, refer to [9].

4. Dijkstra's Graph Algorithm

All shortest path based graph algorithms follow a sequence of predefined graph-search procedures to ensure the shortest paths are found (if any) within the minimum period of time. In this paper, Dijkstra's algorithm is selected for illustration purpose. Refer the pseudo-code in Listing 1.

Given a graph $G = [V, E]$, where $V[G]$ denotes the set of vertices and $W[G]$ denotes the set of edge-weights. We use s to denote the source-vertex, for any two adjacently-connected vertices, $v = Adj[u]$ or $u = Adj[v]$. $d[u]$ denotes total-path-length from s to u, whereas $d[v]$ denotes total-path-length from s to v. Given $w(u, v)$ denoted the edge-weights from u to v, then $d[v] = d[u] + w(u, v)$. Specifically for shortest path algorithm, S is the set used to hold the nearest vertex from source, $\pi[v]$ is used to hold the precedent-vertex of v. Upon complete execution of algorithm, the shortest path from s to v can be traced by dereference $\pi[v]$ backward to source, and the total-path-length is given by $d[v]$.

Dijkstra's algorithm begins with source-node, where all nodes adjacent to source will be scanned and the edge-weights inserted into priority queue. Then the edge with minimum-weight will be extracted from priority queue, and the node in which this minimum-weight-edge heading-to becomes active. The same scanning mechanism continues at this current active node, with insertion to priority queue if there is no identical entry in queue, or relaxation (decrease-key) if identical entry already exists in queue. The procedure continues until all nodes have been visited.

```
DIJKSTRA(G, w, s){
    for (each vertex $v \in V[G]$ and $v \neq s$){
            $d[v] \leftarrow \infty$
            $\pi[v] \leftarrow$ NIL
    }
    $d[s] \leftarrow 0$
    INSERT(Q, s, d[s])
    $S \leftarrow \emptyset$
    do{
        $(u, d[u]) \leftarrow$ EXTRACT-MIN(Q)
        $S \leftarrow S \cup \{u\}$
        for (each vertex $v \in Adj[u]$){
            if ($d[v] = \infty$){
                $d[v] \leftarrow d[u] + w(u, v)$
                $\pi[v] \leftarrow u$
                INSERT(Q, v, d[v])
            }
            elseif ($d[v] > d[u] + w(u, v)$){
                $d[v] \leftarrow d[u] + w(u, v)$
                $\pi[v] \leftarrow u$
                DECREASE-KEY(Q, v, d[v])
            }
        }
    }(while $Q \neq \emptyset$)
}
```

Listing 1. Pseudo-code of Dijkstra's Algorithm

```
DECREASE-KEY (Q, x, new(x_priority))
➔ Find element x in queue, Q;
   If (new(x_priority) dominates old(x_priority))
       x_priority ⟸ new(x_priority).
```

Figure 5. Decrease-Key function

Notice that besides the basic priority queue operations in Figure 1, the graph algorithm needs one additional function, the DECREASE-KEY. The DECREASE-KEY function is very important because it is necessarily for any graph algorithm to perform **RELAXATION** (see [1]). Recall each element ***x*** in queue ***Q*** comes with its ID (here, denoted as ***x_ID***) and its associate-priority (here, denoted as ***x_priority***). Refer to Figure 5, when it comes to a condition where the associate-priority, ***x_priority*** of an element ***x*** needs update, the DECREASE-KEY is invoked to find that element ***x*** in queue; if the new-associate-priority-of-x, ***new*(*x_priority*)** possesses higher priority than the old-associate-priority-of-x, ***old*(*x_priority*)** (the one exist in queue), then replace that associate-priority-of-x. Such property is called **dominancy**, and it is the key to understand graph algorithms.

```
DIJKSTRA_MODIFIED(G, w, s){
    for (each vertex v ϵ V[G]){
        d[v] 🡐 ∞
        π[v] 🡐 NIL
    }
    d[s] 🡐 0
    INSERT(Q, s, d[s])
    S 🡐 Ø
    do{
        do{
            (u, temp) 🡐 EXTRACT-MIN(Q)
        }(while d[u] ≠ temp)
        S 🡐 S U {u}
        for (each vertex v ϵ Adj[u]){
                if (d[v] = ∞){
                    d[v] 🡐 d[u] + w(u, v)
                    π[v] 🡐 u
                    INSERT(Q, v, d[v])
                }
                elseif (d[v] > d[u] + w(u, v)){
                    d[v] 🡐 d[u] + w(u, v)
                    π[v] 🡐 u
                    INSERT(Q, v, d[v])
                }
        }
    }( while Q not empty)
}
```

Listing 2. Modified version of Dijkstra's Algorithm

All software implementation of priority queue in heap (Binary-Heap, Binomial-Heap, and Fibonacci-Heap) could easily incorporate all three functions needed in finding the shortest path: INSERT, EXTRACT-MIN, and DECREASE-KEY. However, neither hardware implementation of priority queue ever provides DECREASE-KEY function; this function is possible to be implemented but often ignored due to additional hardware resources required. The design of SAPQ also does not incorporate DECREASE-KEY function. Hence, modification on the targeted Dijkstra's graph algorithm is needed, so that the algorithm does not invoke DECREASE-KEY directly for relaxation, but indirectly perform relaxation using the only available **INSERT(*Q, x*)** and **EXTRACT-MIN(*Q*)**.

The modified version of Dijkstra's algorithm is presented in Listing 2, which only invoked INSERT and EXTRACT-MIN functions. The sequence of execution remains except (i) when supposed DECREASE-KEY is needed, INSERT is invoked instead; this causes the queue size to actually grow by one, and there is one entry in queue which is no longer valid. (ii) During EXTRACT-MIN, the returned queue-entry will be checked for validity, until a valid entry is returned. Such modification retains the relaxation property of graph algorithm, except the priority queue size at some instance might grow larger than algorithm in Listing 1. Anyway, such exception does not have profound effect on SAPQ since the operation run-time is constant, unlike ISPQ, BHPQ, or other heaps which the operation run-time depends on queue-size.

5. Embedded System Implementation

Targeted for embedded system implementation, SAPQ will serve as co-processor, to off-load the recursive priority queue access and maintenance from the host processor. Our implementation, assuming a general purpose processor, NIOS, to serve as host processor, executing (modified) Dijkstra's algorithm, with the priority queue access and maintenance fully handled by the SAPQ. The whole system is prototyped on a FPGA as illustrated in Figure 6. The Avalon Interface Unit is designed to handle the communication protocol between host processor and SAPQ. The embedded system could be deployed to handle all kinds of application which utilize priority queue. The Dijkstra's algorithm executed by the host processor can be replaced by other algorithms such as discrete-event simulation, global-positioning-system, mobile navigating, etc.

Due to the limitation of Avalon bus bandwidth, which is 32-bit width, the SAPQ does not actually execute at its 64-bit interfacing capability. An optimal SAPQ could be achieved by having it implemented in higher-end communication bridge, such as PCI-64 as the system bus.

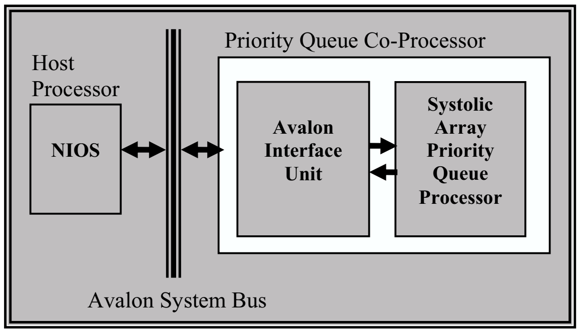
Host Processor
NIOS
Priority Queue Co-Processor
Avalon Interface Unit
Systolic Array Priority Queue Processor
Avalon System Bus

Figure 6. Embedded System on FPGA

6. Result

SAPQ with queue-size of 200 entries is implemented on the said embedded system; this queue-size is constrained by the logic resources of this particular FPGA. Several software priority queues (ISPQ, BHPQ, and FHPQ) are used to compare the performance of SAPQ. Having the SAPQ actually running at lower than optimal speed (50 MHz compared to the maximum allowed is 240 MHz), narrower bus bandwidth (32-bit compared to allowed 64-bit), and high redundancy in cycles per operation incurred by the host processor (44 cycles for INSERT and 52 cycles for EXTRACT; compared to actually 3 cycles designed for

INSERT and 3 cycles designed for EXTRACT), the gain achieved in terms of number of cycles spent per operation is still significant.

(Queue Size = n)	Resource Utility		Run Time Complexity, n	
	# of Processor	# of Memory Storage	INSERT	EXT-MIN
Systolic Array Priority Queue, SAPQ	O(n)	NIL	O(1)	O(1)
Binary-Heap Priority Queue, BHPQ	O(1)	O(n)	O(lg n)	O(lg n)
Insertion-Sort Priority Queue, ISPQ	O(1)	O(n)	O(n)	O(1)
Fibonacci-Heap Priority Queue, FHPQ	O(1)	O(n)	O(1)	O(lg n)*

Table 1: Comparisons in terms of run-time complexity.

(Queue size limited to n=200)	WORST CASE (number of cycles)		SPEED UP GAIN (Achieved by SAPQ)	
	INSERT	EXT-MIN	INSERT	EXT-MIN
Systolic Array Priority Queue, SAPQ	44	52	1.00	1.00
Binary-Heap Priority Queue, BHPQ	753	1185	17.11	22.79
Insertion-Sort Priority Queue, ISPQ	13674	65	310.77	1.25
Fibonacci-Heap Priority Queue, FHPQ	799	44262	18.16	851.19

Table 2: Comparison in terms of processor cycles.

Refer to Table 1, the performance of priority queue depends on the queue-size, **n**. In Table 2, the worst case INSERT is for inserting entries so that the queue is full. The worst case EXT-MIN is for extracting the minimum entry when the queue is full.

Compared to BHPQ, the speed up gain achieved by SAPQ is more than expected. Theoretically, one could expect **lg n** speed up (which in this case n = 200, $\log_2 200 = 7.6$) by SAPQ; this real implementation achieved 17 times and 22 times worst-case speed up for INSERT and EXTRACT operation. This is because software implementation suffers severe memory communication overhead, where tremendous cycles are spent to access heap data structure stored in memory; whereas SAPQ, having all the elements stored in registers, such communication drawback is eliminated.

* For FHPQ, the O(lg n) for EXT-MIN is worst-case amortized-time, refer [1] for amortized analysis.

Next, compared to ISPQ, SAPQ obtained 310 speed gains for worst-case INSERT and 1.25 gains for EXTRACT. Notice the 310 times speed gain is also greater than theoretical expectation; it is also due to the advantage of hardware implementation which eliminates the memory communication overhead.

Lastly, comparison is made between the theoretically most efficient priority queue, the FHPQ with our SAPQ; the result is very impressive. Both FHPQ and SAPQ claims O(***1***) run-time complexity for INSERT, this implementation shows 18 times gain achieved using SAPQ because constantly large cycles are spent by FHPQ to handle a bunch of pointer manipulation. Similarly, for EXTRACT operation, the speed-up gain achieved by SAPQ is over 851 times.

The above reported worst case speed-up gain could be more if larger SAPQ were to be implemented (i.e. queue-size of 1000). The main drawback of Systolic Array Priority Queue Processor is the logic resources consumed, when other software implementation of software priority queues only take space in random access memory. Anyway, for real-time applications where the speed is top-priority, the drawback in logic consumption is a worthy trade-off.

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Clifford Stein, Introduction To Algorithms, 2nd Edition, The MIT Press, McGraw-Hill Book Company, 2001.

[2] S. S. Skiena, The Algorithm Design Manual, Springer-Verlag, New York, 1997

[3] Ch'ng Heng Sun, Mohamed Khalil Hani, "Design of a Graph Processor for VLSI Routing", Master Research Proposal, FEB. 2006.

[4] D. Picker and R. Fellman, "A VLSI Priority Packet Queue with Inheritance and Overwrite," IEEE Trans. Very Large Scale Integration Systems, vol. 3, no. 2, pp. 245-252, June 1995.

[5] R. Brown, "Calendar Queues: A Fast O(1) Priority Queue Implementation for the Simulation Event Set Problem," Comm. ACM, vol. 31, no. 10, pp. 1220-1227, Oct 1988.

[6] J. Chao, "A Novel Architecture for Queue Management in the ATM Network," IEEE J. Selected Areas in Comm., vol. 9, no. 7, pp. 1110-1118, Sept. 1991.

[7] K. Toda, K. Nishida, E. Takahashi, N. Michell, and Y. Tamaguchi, "Design and Implementation of a Priority Forwarding Router Chip for Real-Time Interconnect Networks," Int'l J. Mini and Microcomputers, vol. 17, no. 1, pp. 42-51, 1995.

[8] S.W. Moon, J. Rexford, K.G. Shin, "Scalable Hardware Priority Queue Architectures for High-Speed Packet Swicthes", IEEE Trans. On Computers, vol. 49, no. 11, Nov. 2000.

[9] Ch'ng Heng Sun, Mohamed Khalil Hani, "Systolic Array Priority Queue Processor", Research Report, APR. 2006.