Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks

Aggelos Ioannou[†] and Manolis Katevenis[†]

Institute of Computer Science (ICS), Foundation for Research & Technology – Hellas (FORTH)
Science and Technology Park of Crete, P.O.Box 1385, Heraklion, Crete, GR 711 10 Greece
http://archvlsi.ics.forth.gr/muqpro/heapMgt.html , ioannou@ics.forth.gr , katevenis@ics.forth.gr
Tel: +30 (81) 391660, fax: 391661

Abstract – **Quality-of-Service (QoS) guarantees in networks are increasingly based on per-flow queueing and sophisticated scheduling. Most advanced scheduling algorithms rely on a common computational primitive: priority queues. Large priority queues are built using calendar queue or heap data structures. To support advanced scheduling at OC-192 (10 Gbps) rates and above, pipelined management of the priority queue is needed. We present a pipelined heap manager that we have designed as a core integratable into ASIC's, in synthesizable Verilog form. We discuss how to use it in switches and routers, its advantages over calendar queues, and we present cost-performance tradeoffs. Our design can be configured to any heap size. We have verified and synthesized our design and present cost and performance analysis information.**

KEYWORDS: *high speed network scheduling, weighted round robin, weighted fair queueing, priority queue, pipelined hardware heap, synthesizable core.*

I INTRODUCTION

The speed of networks is increasing at a dramatic pace. Significant advances also occur in network architecture, and in particular in the provision of quality of service (QoS) guarantees. Switches and routers increasingly rely on specialized hardware to provide the desired high throughput and advanced QoS. Such supporting hardware becomes feasible and economical owing to the advances in semiconductor technology. To be able to provide top-level QoS guarantees, network switches and routers increasingly rely on *per-flow queueing* and *advanced scheduling* [14]. The topic of this paper is hardware support for advanced scheduling.

Per-flow queueing refers to the architecture where the packets contending for and awaiting transmission on a given output link are kept in multiple queues, thus providing isolation between flows. A scheduler must then serve these queues in an order that fairly allocates the available throughput to the active flows. Commercial switches and routers currently have multiple queues per output, but their number is limited (a few tens), so their schedulers are relatively simple. When more queues are desired, the hardware architecture has to be adapted accordingly. Managing many thousands of queues at high speed is feasible, today, using modern VLSI technology [13].

This paper deals with the next problem: implementing sophisticated scheduling algorithms at high speed, when there are many thousands of contending flows. Section II presents an overview of various advanced scheduling algorithms. They all rely on a common computational primitive for their most time-consuming operation: finding the minimum (or maximum) among a large number of values. Previous work on implementing this primitive at high speed is reviewed in section II.C. However, for OC-192 (10 Gbps) and higher rates, and for packets as short as about 40 bytes, even higher operation rate is needed. To achieve such higher rates, pipelining must be used.

This paper presents a pipelined heap manager that we have designed in the form of a core, integratable into ASIC's. Pipelining the heap operations requires some modifications to the normal (software) heap algorithms, as described in section III. Section IV presents cost-performance tradeoffs. Section V describes our implementation, which is in synthesizable Verilog form. The ASIC core that we have designed is configurable to any size of priority queue. A new operation can be issued in every clock cycle, except that an insert operation or an idle cycle is needed between two successive delete operations.

II PRIORITY QUEUES FOR ADVANCED SCHEDULING

Many advanced scheduling algorithms have been proposed; good overviews appear in [17] and [12, chapter 9]. Priorities is a first, important mechanism; usually a few levels of priority suffice, so this mechanism is easy to implement. Aggregation (hierarchical scheduling) is a second mechanism: first choose among a number of flow aggregates, then choose a flow within the given aggregate [1]. Some levels of the hierarchy contain few aggregates, while others may contain thousands of flows; this paper concerns the latter levels. The hardest scheduling disciplines are those belonging to the weighted round robin family; we review these, next.

[†] also with the Department of Computer Science, University of Crete, Heraklion, Crete, Greece.

A The Weighted Round Robin Family

With weighted round robin scheduling a scheduler must serve the active flows in an order such that the service received by each active flow in any long enough time interval is in proportion to a weight factor associated with the flow. It is *not* acceptable to visit the flows in plain round robin order, serving each in proportion to its weight, because service times for heavy-weight flows would become clustered together, leading to burstiness and large service time jitter. So, the scheduler will have to operate by keeping track of a "next service time" number for each active flow. In each step, we must find the minimum of these numbers, and then increment it if the flow remains active, or delete it if the flow becomes inactive. When a new packet of an inactive flow arrives, that flow has to be reinserted into the schedule.

Many scheduling algorithms belong to this family. This includes both work-conserving and non-work-conserving disciplines. Other important constituents of a scheduling algorithm such as the mechanism for updating the service time of a served flow, or that of a newly-active one, account for algorithm variants such as the virtual clock algorithm, and the earliest-due-date and rate-controlled disciplines [12, ch.9].

B Priority Queue Implementations

All of the above scheduling algorithms rely on a common computational primitive for their most time-consuming operation: a *priority queue,* i.e. finding the minimum (or maximum) of a given set of numbers. Priority queues with only a few tens of entries or with priority numbers drawn from a small menu of allowable values are easy to implement, e.g. using combinational priority encoder circuits. However, for priority queues with many thousand entries and with values drawn from a large set of allowable numbers, *heap* or *calendar queue* data structures must be used. Other heap-like structures [7] are interesting in software but are not adaptable to high speed hardware implementation.

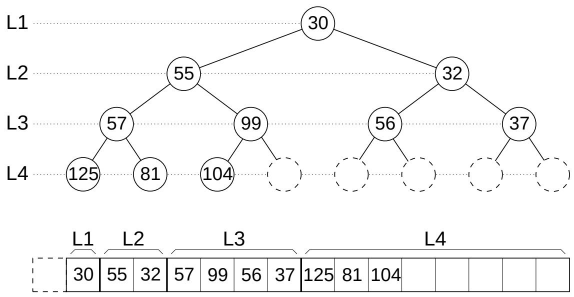

Figure 1: Heap priority queue

Fig. 1 illustrates a heap. It is a binary tree (top), physically stored in a linear array (bottom). Non-empty entries are pushed all the way up and left. The entry in each node is smaller than the entries in its two children (the *heap property*). Insertions are performed at the leftmost empty entry, and then possibly interchanged with their ancestors to re-establish the heap property. The minimum entry is always at the root; to delete it, move the last filled entry to the root, and possibly interchange it with descendants of it that may be smaller. In the worst case, a heap operation takes a number of interchanges equal to the tree height.

A calendar queue [3] is an array of buckets. Entries are placed in the bucket indicated by a linear hash function. The next minimum entry is found by searching in the current bucket, then searching for the next non-empty bucket. Calendar queues have a good average performance, but in the short-term, some operations may be quite expensive.

C Related Work

For small priority queues (a few tens of entries) or for special cases such as plain round robin or round robin with only a small set of weight factors, simple implementations work effectively [16] [11]. Priority queues with up to hundreds of entries using specialized hardware were reported in [10] [4]; however, they do not outperform our pipelined heap manager, while their cost is higher.

For priority queues with many thousands of entries, calendar queues are a viable alternative. In high-speed switches and routers the delay of resizing the calendar queue –as in [3]– is usually unacceptable, so a large size is chosen from the beginning. However, the large size creates long sequences of empty buckets, thus requiring a mechanism to quickly search for the next non-empty bucket [8] [5]. No specific implementations of calendar queues at the performance range considered in this paper have been reported in the literature. However it is hard to give to calendar queues a deterministic response time like the one featured by the pipelined heap, while their cost is higher, because rehashing or linked lists are needed to handle collisions. Also, in order to be efficient, they use significantly more memory , which is the dominant cost at large sizes.

Finally, concerning heap management at high speed, we had studied how fast it can be performed using a hardware FSM manager with the heap stored in one or two off-chip SRAM's [15]. In the present paper, we look at higher-speed heap management, using pipelining. As far as we know, nobody else had looked at pipelined heap management before this work, while a parallel and independent study appeared in [2]. In that paper, Bhagwan and Lin introduce a variant of the conventional heap, which they call *P-heap.* However the P-heap has two disadvantages relative to our architecture. First, the issue rate of insert operations cannot exceed that of consecutive deletes, while we achieve *twice* this speed. Second, the forest-of-heaps optimization (section III.B) is not possible. Regarding pipeline structure, Bhagwan & Lin use a single and long clock cycle to fit all *three* basic operations (read-compare-write) needed at each tree level. Compared to our model that uses a short clock cycle to fit only one basic operation, their design –which uses no pipeline bypasses– would issue a new operation every *six* (6) short cycles, compared to 1 or 2 of our implementation.

III PIPELINING THE HEAP MANAGEMENT

Section II showed why heap data structures play a central role in the implementation of advanced scheduling algorithms. When the entries of a large heap are stored in off-chip memory, or even with the top few levels of the heap in on-chip memory, using off-chip memory only for the bottom (larger) levels, the desire to minimize pin cost entails little parallelism in accessing them. For highest performance, the entire heap can be on-chip, so as to use parallelism in accessing all its levels, as described in this section. Such highest performance –up to 1 operation per clock cycle– will be needed e.g. in OC-192 line cards. An OC-192 input line card must handle an incoming 10 Gbit/s stream plus an outgoing (to the switching fabric) stream of 15 to 30 Gbit/s. At 40 Gbps, for packets as short as about 40 bytes, the packet rate is 125 M packets/s; each packet may generate one heap operation, hence the need for heap performance in excess of 100 M operations/s. A wide spectrum of intermediate solutions exist too, as discussed in section IV on cost-performance tradeoffs.

A Heap Algorithms for Pipelining

Fig. 2 illustrates the basic ideas of pipelined heap management. Each level of the heap is stored in a separate physical memory, and managed by a dedicated controller stage. The external world only interfaces to stage 1. The operations provided are *i) insert* a new entry into the heap (on packet arrival, when the flow becomes non-idle); *ii) deleteMin:* read and delete the minimum entry i.e. the root (on packet departure, when the flow becomes idle); and *iii) replaceMin:* replace the minimum with a new entry that has a higher value (on packet departure, when the flow remains non-idle). When a stage is requested to perform an operation, it performs the operation on the appropriate node at its level, and then it recursively asks the level below to also perform an induced operation. For levels 2 and below, the node index, i, must also be specified. Each stage is thus able to process a new operation as soon as it has completed the previous operation at its own level only.

The *replace* operation is the easiest to understand. In Fig. 2, the given $\mathrm{arg1}$ must replace the root at level 1. Stage 1 reads its two children from L2, to determine which of the three values is the new minimum, to be written into L1; if one of the ex-children was the minimum, the given $\mathrm{arg1}$ must

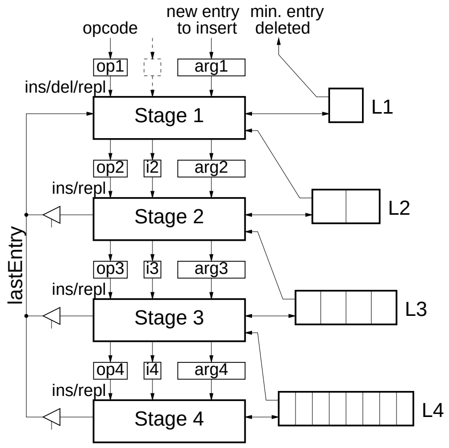

Figure 2: Simplified block diagram of the pipeline

now replace that child, giving rise to a *replace* operation for stage 2, and so on.

The *delete* operation is similar to replace. The $\mathrm{arg1}$ is now either the rightmost non-empty entry of the bottom-most non-empty level (which is then deleted), or, when multiple operations are in progress in various pipeline stages, it comes from the youngest-in-progress *insert* (which is then aborted). The $\mathrm{lastEntry}$ bus is now used to provide $\mathrm{arg1}$.

The traditional *insert* algorithm needs to be modified [9] [15]. Instead of inserting the new entry at the bottom, it is inserted at the root, in order for all the operations to proceed top-to-bottom. Recursive repositionings are then performed to *the proper of the two* sub-heaps. By properly steering –left or right sub-heap– this chain of insertions at each level, we can ensure that the last insertion will be guided to occur at precisely the heap node next to the previously-last entry.

Each operation on a node i, in each stage of Fig. 2, takes 3 clock cycles: *i)* read from memory; *ii)* compare two or three values to find the minimum; *iii)* write this minimum into the memory of this stage. Using such an execution pattern, operations ripple down the pipeline at the rate of one stage every 3 clocks, allowing an operation initiation rate no higher than 1 every 3 cycles. We can improve on this rate by overlapping the operation of stages. In this way an operation can start working on consecutive levels, before the work to be done on previous levels has completed. We can thus end up with a ripple-down rate of one stage every cycle, requiring a read throughput of 4 values per cycle in each memory, plus an additional write throughput of 1 entry per cycle. Cost-performance tradeoffs are further analyzed in section IV.

B Managing a Forest of Multiple Heaps

In a system that employees hierarchical scheduling (section II), there are multiple sets (aggregates) of flows. At the second and lower hierarchy levels, we want to choose a flow within a given aggregate. When priority queues are used for this latter choice, we need a manager for a forest of heaps –one heap per aggregate. Our pipelined heap manager can be conveniently used to manage such a forest. Referring to Fig. 2, it suffices to store all the heaps "in parallel", in the memories L1, L2, L3, ..., and to provide an index $i1$ to the first stage (dashed lines), identifying which heap in the forest the requested operation refers to. Furthermore, for a given maximum number of flows in the system, great memory savings can be achieved for the large memories near the leaves [6].

IV Cost-Performance Tradeoffs

A wide range of cost-performance tradeoffs exists for pipelined heap managers. The highest performance (unless one goes to superscalar organizations) is for operations to ripple down the heap at one level per clock cycle, and for new operations to also enter the heap at that rate. Analysis however showed that this requires 2-port memory blocks that are 4-entry wide, plus expensive global bypasses [6]. This high-cost, high-performance option appears in line *(i)* of Table 1. To have a concrete notion of memory width in mind, in our example implementation (section V) each heap entry is 32 bits wide. To avoid global bypasses, which require expensive datapaths and may slow down the clock cycle, delete (or replace) operations have to consume 2 cycles each when immediately following one another, as noted in line *(ii)*. In many cases, this will be an insignificant performance penalty, because one can often arrange for one or more insertions to be interposed between deletions, in which case the issue rate is still one operation per clock cycle.

Table 1 : Alternatives with on-chip SRAM

LINE	COST			$\frac{1}{Perf.}$ (CPI)	
	SRAM				
	# Ports	Width (entr.)	Bypass Paths	cc per DEL	cc Per INS
i	2	4	global	1	1
ii	2	4	local	1 or 2	1
iii	1	4	local	2	2
iv	1	2	global	2	2
v	1	2	local	3	2
vi	1	1	local	4	2

To further reduce cost, single-port memories can be used. The corresponding speed can be seen at line *(iii)* and below. If we go further on and reduce memory width, mainly economizing on power consumption, we can have the performance of line *(v)*. Line *(iv)* gives an alternative with global bypasses, which however opposes to cost reduction. Finally line *(vi)* concerns single-entry-wide memories.

Another option studied, was to place the last one or two levels of the heap (the largest ones) in a *single* (one port) off-chip SRAM in order to economize in silicon area. We consider off-chip memories of width 1 or 2 heap entries[1]. The read-compare-write loop for delete operations now becomes longer. Thus, the issue rate is as presented in Table 2.

Table 2 : Alternatives With Off-chip SRAM

LINE	COST		$\frac{1}{Perf.}$ (CPI)	
	Off-Chip SRAM			
	Width (entries)	Levels contained	cc per DEL	cc Per INS
i	2	1	5	2
ii	1	1	6	2
iii	2	2	5	4
iv	1	2	6	4

V Implementation

We have designed a pipelined heap manager as a core integratable into ASIC's, in synthesizable Verilog form. We chose to implement the version appearing in line *(ii)* of table 1 (section IV). Replace operations are not supported (but can be added easily), because they are often implemented as split delete-insert transactions, with some delay between the delete and the insert operations.

In order to verify our design, we wrote three models of a heap, of increasing level of abstraction, and we simulated them in parallel with the Verilog design, so that each higher-level model checked the correctness of the next level, down to the actual design. We have verified the design with many different operation sequences, activating all existing bypass paths. Test patterns of tens of thousands of operations were used, in order to test all levels of the heap, also reaching saturation conditions.

In an example implementation that we have written, each heap entry consists of an 18-bit priority value and a 14-bit flow identifier, for a total of 32 bits per entry. Each pipeline stage stores the entries of its heap level in four 32-bit two-port SRAM blocks. We have processed the design through Synopsys to get area and performance information. For a 16K entry heap, the largest SRAM blocks are $2K \times 32$. Most of the area for the design is consumed by the unavoidable on-chip memory. The datapath and control of one (general) pipeline stage have a complexity of about 5.5K gates[2] plus 500 bits

[1] We assumed zero-bus-turnaround (ZBT – IDT trademark; see http://www.micron.com/mti/msp/html/zbtds.html) off-chip memory, which operates with the same clock as the rest of the heap

[2] simple 2-input NAND/NOR gates

of flip-flops/registers. For our example implementation, the resulting complexity is about 80K gates, 7K register bits, and 0.5M memory bits. The approximate area consumed, for a 0.18-micron CMOS ASIC library, is 14.5 mm^2 in total, with 4.3 mm^2 concerning datapath.

The clock frequency is approximately 180MHz for a heap of 16K entries, in a conservative 0.18-micron technology. Using more efficient technology than the low-power-low-speed one we used, we estimate that the clock frequency would reach 250 to 300MHz. With a clock speed of 200MHz, this heap provides a throughput of 200 Mega Operations per Second (Mops) (100 Mops for consecutive delete operations with no interposed insertions). Even for packets as small as 40 bytes, 200 Mops translates to a rate of about 64Gbps. For more information on our implementation see [6].

CONCLUSIONS : We have designed a pipelined heap manager, thus demonstrating the feasibility of large priority queues with throughput rates above 100 million operations per second, at reasonable cost. The feasibility of priority queues with many thousands of entries in this throughput range has important implications for high speed networks and the future internet. Many sophisticated algorithms for providing top-level quality-of-service guarantees rely on per-flow queueing and priority-queue-based schedulers. Thus, we have demonstrated the feasibility of these algorithms, at reasonable cost, at OC-192 (10 Gbps) and higher rates.

ACKNOWLEDGMENTS : We would like to thank all those who helped us, and in particular George Kornaros and Dionisios Pnevmatikatos. We also thank Europractice and the University of Crete for providing many of the CAD tools used, and the Greek General Secretariat for Research & Technology for the funding provided.

REFERENCES

[1] J. Bennett, H. Zhang: "Hierarchical packet fair queueing algorithms", *IEEE/ACM Trans. on Networking,* vol. 5, no. 5, Oct. 1997, pp. 675-689.

[2] R. Bhagwan, B. Lin: "Fast and scalable priority queue architecture for high-speed network switches" *IEEE Infocom 2000 Conference,* 26-30 March 2000, Tel Aviv, Israel; http://www.ieee-infocom.org/2000/papers/565.ps

[3] R. Brown: "Calendar Queues: a fast O(1) priority queue implementation for the simulation event set problem", *Commun. of the ACM,* vol. 31, no. 10, Oct. 1988, pp. 1220-1227.

[4] H. J. Chao: "A novel architecture for queue management in the ATM network", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 9, no. 7, Sep. 1991, pp. 1110-1118.

[5] H. J. Chao, Y. Jeng, X. Guo, C. Lam: "Design of packet-fair queueing schedulers using a RAM-based searching engine", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 17, no. 6, June 1999, pp. 1105-1126.

[6] A. Ioannou: “An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks”, Master of Science Thesis, University of Crete, Greece; *Technical Report FORTH-ICS/TR-278,* Institute of Computer Science, FORTH, Heraklio, Crete, Greece, November 2000; http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

[7] D. Jones: "An empirical comparison of priority-queue and event-set implementations", *Commun. of the ACM,* vol. 29, no. 4, Apr. 1986, pp. 300-311.

[8] M. Katevenis: "Fast switching and fair control of congested flow in broad-band networks", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 5, no. 8, Oct. 1987, pp. 1315-1326.

[9] M. Katevenis, lectures on heap management, Fall 1997.

[10] M. Katevenis, S. Sidiropoulos, C. Courcoubetis: "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 9, no. 8, Oct. 1991, pp. 1265-1279.

[11] M. Katevenis, D. Serpanos, E. Markatos: “Multi-queue management and scheduling for improved QoS in communication networks”, *Proceedings of EMMSEC'97* (European Multimedia Microprocessor Systems and Electronic Commerce Conference), Florence, Italy, Nov. 1997, pp. 906-913; http://archvlsi.ics.forth.gr/html_papers/ EMMSEC97/paper.html

[12] S. Keshav: "An engineering approach to computer networking", *Addison Wesley,* 1997, ISBN 0-201-63442-2.

[13] A. Nikologiannis, M. Katevenis: “Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques”, *Proc. IEEE Int. Conf. on Communications (ICC'2001),* Helsinki, Finland, June 2001; http://archvlsi.ics.forth.gr/muqpro/queueMgt.html

[14] V. Kumar, T. Lakshman, D. Stiliadis: "Beyond best effort: Router architectures for the differentiated services of tomorrow's internet", *IEEE Communications Magazine,* May 1998, pp. 152-164.

[15] I. Mavroidis: "Heap management in hardware", *Technical Report FORTH-ICS/TR-222,* Institute of Computer Science, FORTH, Crete, GR; http://archvlsi.ics.forth.gr/muqpro/heapMgt. html

[16] D. Stephens, J. Bennett, H. Zhang: "Implementing scheduling algorithms in high-speed networks", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 17, no. 6, June 1999, pp. 1145-1158. http://www.cs.cmu.edu/People/hzhang/ publications.html

[17] H. Zhang: "Service disciplines for guaranteed performance in packet switching networks", *Proceedings of the IEEE,* vol. 83, no. 10, Oct. 1995, pp. 1374-1396.