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
{ioannou, katevenis}@ics.forth.gr Tel: +30 (81) 391664, fax: 391661

http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

January 2001

ABSTRACT: Per-flow queueing with sophisticated scheduling is a candidate architecture for networks that provide advanced Quality-of-Service (QoS) guarantees. The hardest and most interesting scheduling algorithms rely on a common computational primitive: priority queues. Large priority queues are built using calendar queue or heap data structures. To support such scheduling for a large number of flows at OC-192 (10 Gbps) rates and beyond, pipelined management of the priority queue is needed. We present heap management algorithms that can be gracefully pipelined; they constitute modifications of the traditional ones. We discuss how to use a pipelined heap manager in switches and routers, its advantages over calendar queues, its structure and its cost-performance tradeoffs. The design can be configured to any heap size, and, using 2-port 4-wide SRAM's, it can support initiating a new operation on every clock cycle, except that an insert operation or one idle (bubble) cycle is needed between two successive delete operations. We present a pipelined heap manager implemented in synthesizable Verilog form, as a core integratable into ASIC's. We have verified our design by co-simulating it with three heap models of varying abstraction. We have also performed synthesis, and present cost and performance analysis information. For a 16K entry example in 0.18-micron CMOS technology, silicon area is approximately $15mm^2$ and performance is a few hundreds of millions operations per second.

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

*the authors are also with the Department of Computer Science, University of Crete, Heraklion, Crete, Greece.

1 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 need *per-flow queueing* and *advanced scheduling* [Kumar98]. The topic of this paper is hardware support for advanced scheduling when the number of flows is high.

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. In the alternative –single-queue systems– the service discipline is necessarily first-come-first-served (FCFS), which lacks fairness and isolation among well-behaved and ill-behaved flows. A partial solution is to use FCFS but rely on traffic regulation at the sources, based on service contracts (admission control) or on end-to-end flow control protocols (like TCP). While these may be able to achieve fair allocation of throughput in the long term, they suffer from short-term inefficiencies: when new flows request a share of the throughput, there is a delay in throttling down old flows, and, conversely, when new flows request the use of idle resources, there is a delay in allowing them to do so. In modern high-throughput networks, these round-trip delays –inherent in any control system– correspond to an increasingly large number of packets.

To provide real isolation between flows, the packets of each flow must wait in separate queues, and a scheduler must serve the queues in an order that fairly allocates the available throughput to the active flows. Note that fairness does not necessarily mean equality –service differentiation, in a controlled manner, is a requirement for the scheduler. Commercial switches and routers already have multiple queues per output, but the number of such queues is small (a few tens or less), and the scheduler that manages them is relatively simple (e.g. plain priorities).

This paper shows that sophisticated scheduling among many thousands of queues at very high speed is technically feasible, at a reasonable implementation cost. Given that it is also feasible to manage many thousand of queues in DRAM buffer memory at OC-192 rates [Niko01], we conclude that fine-granularity per-flow queueing and scheduling is technically feasible even at very high speeds.

Section 2 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 2.3. For large numbers of widely dispersed values, priority queues in the form of *heap data structures* are the most efficient representation, providing insert and delete_minimum operations in logarithmic time. For a heap of several thousand entries, this translates into a few tens of accesses to a RAM per heap operation, which yields an operation rate of 5 to 10 million operations per second (Mops) or less [Mavro98]. However, at

OC-192 (10 Gbps) rates and beyond, and for packets as short as about 40 bytes, quite more than 60 Mops will be needed. To achieve such higher rates, the heap operations must be pipelined.

Pipelining the heap operations requires some modifications to the normal (software) heap algorithms, as we proposed in 1997 [Kate97] (see the Technical Report [Mavro98]). This paper presents a pipelined heap manager that we have designed in the form of a core, integratable into ASIC's. Section 3 presents our modified management algorithms for pipelining the heap operations. Then, we explain the pipeline structure and the difficulties that have to be resolved for the rate to reach one heap operation per clock cycle. Reaching such a high rate requires expensive SRAM blocks and bypass paths. A number of alternatives exist that trade off performance to achieve a lower cost; these are analyzed in section 4.

Section 5 describes our implementation, which is in synthesizable Verilog form. The ASIC core that we have designed is configurable to any size of priority queue. With its clock frequency able to reach a few hundreds MHz in 0.18-micron CMOS technology, operation rate reaches one or more hundreds of million of operations per second. More details both on the algorithms and the corresponding implementation can be found in [Ioann00].

The contributions of this paper are: *(i)* it presents heap management algorithms that are appropriate for pipelining; *(ii)* it describes an implementation of a pipelined heap manager and reports on its cost and performance; *(iii)* it analyzes the cost-performance tradeoffs of pipelined heap managers. As far as we know, similar results have not been reported before, except for the independent work [Bhagwan00] which differs from this work as described in section 3.6. The usefulness of our results stems from the fact that pipelined heap managers may become crucial for providing advanced QoS features in high speed networks of the future.

2 Advanced Scheduling using Priority Queues

Section 1 explained the need for per-flow queueing in order to provide advanced QoS in future high speed networks. To be effective, per-flow queueing needs a good scheduler. Many advanced scheduling algorithms have been proposed; good overviews appear in [Zhang95] and [Keshav97, 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 [Bennett97]. 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.

2.1 The Weighted Round Robin Family

Figure 1 intuitively illustrates weighted round robin scheduling. Seven flows, A through G, are shown; four of them, A, C, D, F are currently active (non-empty queues). The 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 its weight factor. 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.

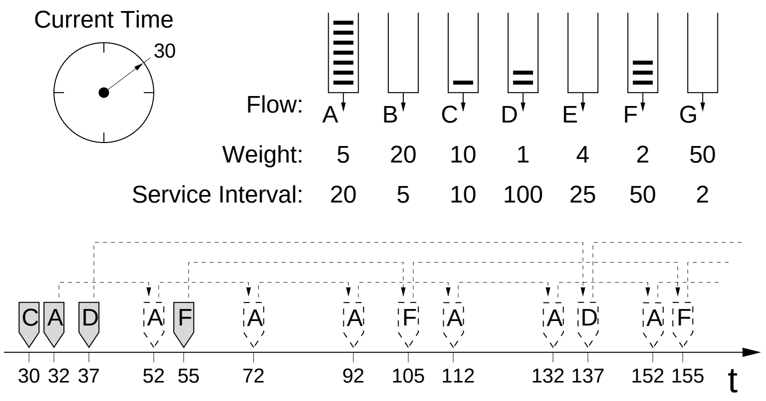

Figure 1: Weighted round robin scheduling

We begin with the active flows "scheduled" to be served in a particular future "time" each: flow A will be served at t=32, C at t=30, D at t=37, and F at t=55. The flow to be served next is the one that has the earliest, i.e. the minimum, scheduled service time. In our example, this is flow C; its queue only has a single packet, so after being served it becomes inactive and it is removed from the schedule. Next, "time" advances to 32 and flow A is served. Flow A remains active after its head packet is transmitted, so it has to be rescheduled. Rescheduling is based on the inverses of the weight factors, which correspond to relative service intervals[1]. The service interval of A is 20, so A is rescheduled to be served next at "time" 32+20=52. (When packet size varies, that size also enters into the calculation of the next service time). The resulting service order is shown in figure 1. As we see, the scheduler operates 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. Work-conserving disciplines always advance the "current time" to the service time of the next active flow, no matter how far in the future that time is; in this way, the active flows absorb all the available network capacity. Non-work-conserving disciplines use a real-time clock. A flow is only served when the real-time clock reaches or exceeds its scheduled service time; when no such flow exists, the transmitter stays idle. These schedulers operate as "regulators", forcing each flow to obey its service contract. Other important constituents of a scheduling algorithm are the way it updates the service time of the flow that was served (e.g. based on the flow's service interval or on some per-packet deadline), and the way it determines the service time of newly-active flows. These issues account for the differences among the weighted fair queueing algorithm and its variants, the virtual clock

[1] in arbitrary units; there is no need for these numbers to add up to any specific value, so they do not have to change when new flows become active or inactive.

algorithm, and the earliest-due-date and rate-controlled disciplines [Keshav97, ch.9].

2.2 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 [Jones86] are interesting in software but are not adaptable to high speed hardware implementation.

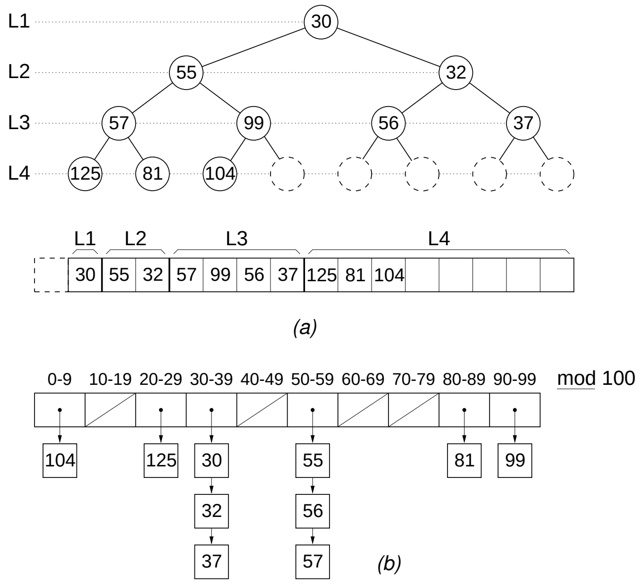

Figure 2: Priority queues: (a) heap; (b) calendar queue

Figure 2(a) 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.

Figure 2(b) illustrates a calendar queue [Brown88]. It 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 when the average is taken over a long sequence of operations. However, in the short-term, some operations may be quite expensive. In [Brown88], the calendar is resized when the queue grows too large or too small; this resizing involves copying of the entire queue. Without resizing, either the linked lists in the buckets may become very long, thus slowing down insertions (if lists are sorted) or searches (if lists are unsorted), or the empty-bucket sequences may become very long, thus requiring special support to search for the next non-empty bucket[2].

2.3 Related Work

Priority queues with up to a few tens of entries can be implemented using combinational circuits. In some cases, one can avoid larger priority queues. In plain round robin, i.e. when all weight factors are equal, scheduling can be done using a circular linked list[3]. In weighted round robin, when the weight factors belong to a small menu of a few possible different values, hierarchical schedulers that use round robin within each weight-value class work well [Stephens99] [KaSM97]. This paper concerns the case when the weight factors are arbitrary.

For hundreds of entries, a weighted round robin scheduler based on content addressable memory (CAM) was proposed in [KaSC91], and a priority queue using per-word logic (sort by shifting) was proposed in [Chao91]. Their performance is similar to that of our pipelined heap manager (1 operation per 1 or 2 clock cycles). However, the cost of these techniques scales poorly to large sizes. At large sizes, memory is the dominant cost (section 5.4); the pipelined heap manager uses SRAM, which costs less than CAM and much less than per-word logic.

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 [Brown88]– is usually unacceptable, so a large size is chosen from the beginning. The large size helps to reduce the average number of entries that hash together into the same bucket; on the other hand, it creates long sequences of empty buckets. Quickly searching for the next non-empty bucket can be done using a hierarchy of bit-masks, where each bit indicates whether all buckets in a certain block of buckets are empty [Kate87] [Chao97] [Chao99]. No specific implementations of calendar queues at the performance range considered in this paper have been reported in the literature, so we cannot make a more specific comparison than the following. Calendar queues can, in principle, be made as fast as our pipelined heap manager, although it is hard to give them a deterministic response time like the one featured by the pipelined heap. From the cost point of view, however, calendar queues have the following disadvantages relative to the pipelined heap. First, calendar queues have to handle collisions of entries hashing into the same bucket, hence the need for rehashing or handling linked lists. Second, each calendar bucket must be large enough to hold at least a flow ID, and, in order to achieve good performance[4], the number of buckets in

[2] In non-work-conserving schedulers searching for the next non-empty bucket is not a problem: since the current time advances with real time, the scheduler only needs to look at one bucket per time slot –empty buckets simply mean that the transmitter should stay idle.

[3] Yet, choosing the proper insertion point is not trivial.

[4] Weighted –rather than plain– round robin is especially useful when weight factors vary widely. In that case, heavy-weight flows have entries clustered near the current time, and light-weight flows have entries (hopefully) distributed throughout the calendar. If the number of buckets is comparable to the number of flows, either heavy-

the calendar should be significantly larger than the number of flows. Hence, a calendar queue needs significantly more memory than an equivalent heap does, and SRAM is the dominant cost in both systems 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 an SRAM block or chip [Mavro98]. In this 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 [Bhagwan00]; that study differs from ours as described in section 3.6.

3 Pipelining the Heap Management

Section 2 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, the desire to minimize pin cost entails little parallelism in accessing them. Under such circumstances, a new heap operation can be initiated every 15 to 50 clock cycles for heaps of sizes 256 to 64K entries, stored in one or two 32-bit external SRAM's [Mavro98][5]. Higher performance can be achieved by maintaining the top few levels of the heap in on-chip memory, using off-chip memory only for the bottom (larger) levels. 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 4 on cost-performance tradeoffs.

3.1 Heap Algorithms for Pipelining

Figure 3 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

weight flows will collide in the same buckets, or the calendar will have to contain multi-epoch entries. These will both require sorted linked lists –thus complicating insertions– and the latter choice will also complicate the search for the next non-empty bucket that contains an entry of the *current epoch*.

[5] These numbers also give an indication on the limits of software heap management, when the heap fits in on-chip cache. When the processor is very fast, the cache SRAM throughput is the limiting factor; then, each heap operation costs 15 to 50 cycles of that on-chip SRAM, as compared to 1 to 4 SRAM cycles in the present design (section 4).

the minimum with a new entry that has a higher value (on packet departure, when the flow remains non-idle).

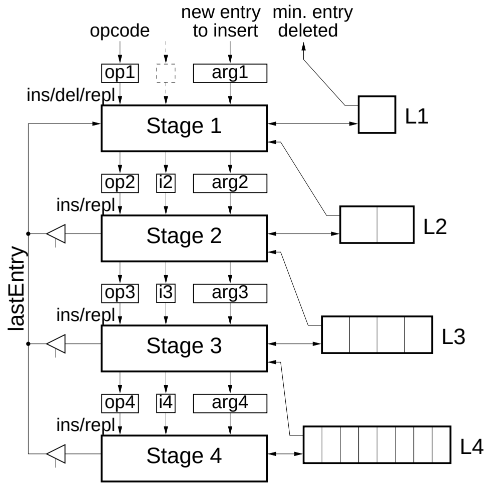

Figure 3: Simplified block diagram of the pipeline

When a stage is requested to perform an operation, it performs the operation on the appropriate node at its level, and then it may request the level below to also perform an induced operation that may be required in order to maintain the heap property. For levels 2 and below, besides specifying the operation and a data argument, the node index, i, must also be specified. When heap operations are performed in this way, each stage (including the I/O stage 1) is ready to process a new operation as soon as it has completed the previous operation at its own level only.

The *replaceMin* operation is the easiest to understand. In figure 3, the given 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 arg1 must now replace that child, giving rise to a *replace* operation for stage 2, and so on.

The *deleteMin* operation is similar to replace. To maintain the heap balanced, the root is deleted by replacing it with the rightmost non-empty entry in the bottom-most non-empty level[6]. A lastEntry bus is used to read that entry, and the corresponding level is notified to delete it. When multiple operations are in progress in various pipeline stages, the real "last" entry may not be the last entry in the last level: the pending value of the youngest-in-progress *insert* operation must be used instead. In this case, the lastEntry bus functions as a bypass path, and the most recent insert operation is then aborted.

The traditional *insert* algorithm needs to be modified as shown in figure 4. In a non-pipelined heap, new entries are inserted at the bottom, after the last non-empty entry (fig. 4(a)); if the new entry is smaller than its parent, it is swapped with that parent, and so on. Such operations

[6] If we did not use the last entry for the root substitution, then the smaller of the two children would be used to fill in the empty root. As this would go on recursively, leaves of the tree would be freed up arbitrarily, spoiling the balance of the tree.

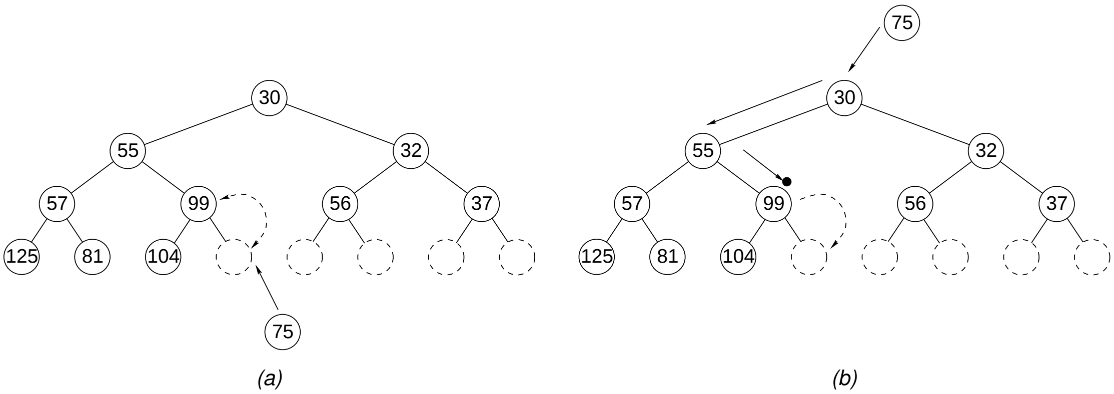

Figure 4: Algorithms for Insert: (a) traditional insertion; (b) modified insertion (top-to-bottom)

would proceed in the wrong direction, in the pipeline of figure 3. In our modified algorithm, new entries are inserted at the root (fig. 4(b)). The new entry and the root are compared; the smaller of the two stays as the new root, and the other one is recursively inserted into *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. The address of that target slot is known at insertion-initiation time: it is equal to the heap occupancy count plus one; the bits of that address, MS to LS, steer insertions left or right at each level. Notice that traditional insertions only proceed through as many levels as required, while our modified insertions traverse all levels; this does not influence throughput, though, in a pipelined heap manager.

3.2 Overlapping the Operation of Successive Stages

Replace (or delete) operations on a node i, in each stage of fig. 3, take 3 clock cycles each: *(i)* read the children of node i from the memory of the stage below; *(ii)* find the minimum among the new value and the two children; *(iii)* write this minimum into the memory of this stage. Insert operations also take 3 cycles per stage: *(i)* read node i from the memory of this stage; *(ii)* compare the value to be inserted with the value read; *(iii)* write the minimum of the two 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. Figure 5 shows replace (or delete) –the hardest operation– in the case of ripple-down rate of one stage per cycle. The operation at level L has to replace value C, in node i_L, by the new value C'. The index i_L of the node to be replaced, as well as the new value C', are deduced from the replace operation at level L-1, and they become known at the end of clock cycle 1, right after the comparison of the new value A' for node i_{L-1} with its two children, B, and C. The comparison of C' with its children, F and G, has to occur in cycle 2 in order to achieve the desired ripple-down rate. For this to be possible, F and G must be read in cycle 1. However, in cycle 1, index i_L is *not yet* known

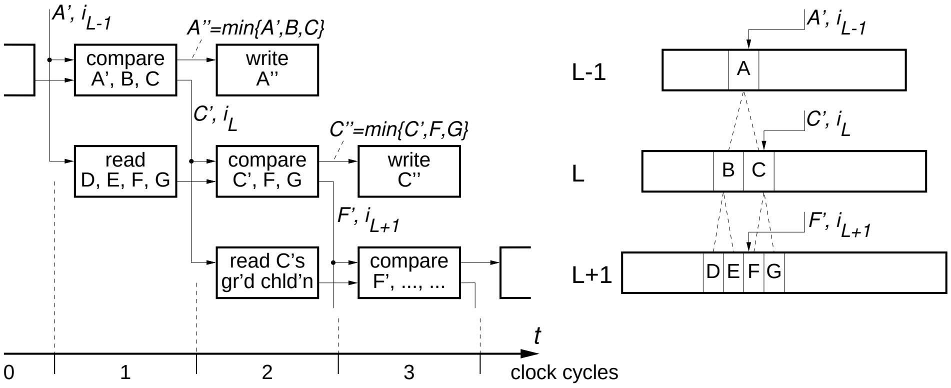

Figure 5: Overlapped stage operation for replace

–only index i_{L-1} is known. Hence, in cycle 1, we are obliged to read *all four* grand children of A (D, E, F, G) given that we do not know yet which one of B or C will need to be replaced; notice that these grand children are stored in consecutive, aligned memory locations, so they can be easily read in parallel from a wide memory. In conclusion, a ripple-down rate of one stage every cycle needs a read throughput of 4 values per cycle in each memory[7]; an additional write throughput of 1 entry per cycle is also needed. Insert operations only need a read throughput of 1, because the insertion path is known in advance. Cost-performance tradeoffs are further analyzed in section 4.

3.3 Inter-Operation Dependencies and Bypasses

Figure 5 only shows a single replace operation, as it ripples down the pipeline stages. When more operations are simultaneously in progress, various dependencies arise. A structural dependence between write-back's of replace and insert operations can be easily resolved by moving the write-back of insert to the fourth cycle (in figure 5, reading at level L is in cycle 0, and writing at the same level is in cycle 3).

Various data dependencies also exist; they can all be resolved using appropriate bypasses. The main data dependency for an insert concerns the reading of each node. An insert can be issued in clock cycle 2 if the previous operation was issued in clock cycle 1. However, that other operation will not have written the new item of stage 1 until cycle 3, while insert tries to read it in cycle 2. Nevertheless, this item is actually needed by the insert only for the comparison, which takes place in cycle 3. At that time the previous operation has already decided the item of stage 1, and it can thus be forwarded to the insert operation. The read however is needed, since two consecutive operations can take different turns as they traverse the tree downwards, and thus they may manipulate distinct sets of elements, so that the results of the read will need to be used instead of the mentioned bypassing. Of course there is a chance that these bypasses

[7]Heap entries contain a priority value and a flow ID. Only the flow ID of the entry to be swapped needs to be read. Thus, flow ID's can be read later than priority values, so as to reduce the read throughput for them, at the expense of complicating pipeline control.

continue as the two operations traverse the tree.

The corresponding data dependency for a delete operation is analogous but harder. The difference is that delete needs to read the entries of level 2, rather than level 1, when it is issued. This makes it dependent on the updating of this level's items, which comes one cycle after that of level 1. So, unlike insert, it cannot immediately follow its preceeding operation, but should rather stall one cycle. However it can immediately follow an insert operation, as this insert will be aborted, thus creating the needed bubble separating the delete from a preceeding operation.

Besides local bypasses from the 2 lower stages (section 5.1), a global bypass was noted in section 3.1: when initiating a deletion, the youngest-in-progress insertion must yield its value through the lastEntry bus.

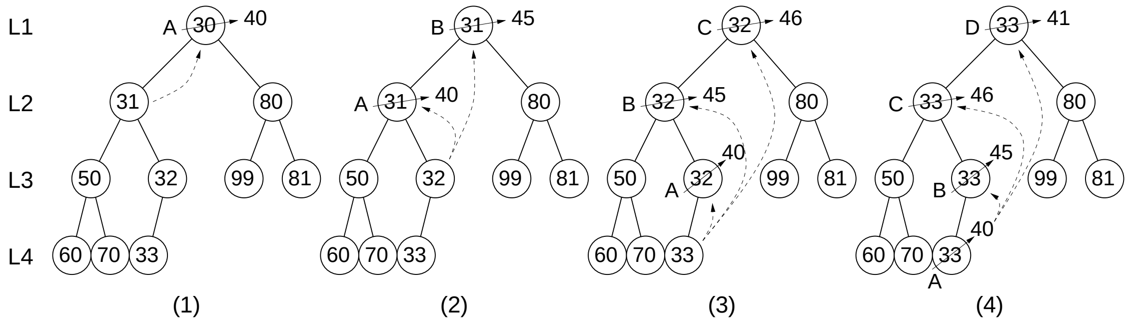

Figure 6: Replace operations in successive stages need global bypasses

The most expensive set of global bypasses is needed for replacements or deletions, when they are allowed to immediately follow one another. Figure 6 shows an example of a heap where the arrangement of values is such that all four successive replacements shown will traverse the same path: 30, 31, 32, 33. In clock cycle (1), in the left part of the figure, operation A replaces the root with 40; this new value is compared to its two children, 31 and 80. In cycle (2), the minimum of the previous three values, 31, has moved up to the root, while operation A is now at L2, replacing the root of the left subtree; a new operation B reads the new minimum, 31, and replaces it with 45. Similarly, in cycle (3) operation C replaces the minimum with 46, and in cycle (4) operation D replaces the new minimum with 41. The correct minimum value that D must read and replace is 33; looking back at the system state in cycle (3), the only place where "33" appears is in L4: the entire chain of operations, in all stages, must be bypassed for the correct value to reach L1. Similarly, in cycle (4), the new value, 41, must be compared to its two children, to decide whether it should fall or stay. Which are its correct children? They are *not* 46 and 80, neither 45 and 80 –these would cause 41 to stay at the root, which is wrong. The correct children are 40 and 80; again, the value 40 needs to be bypassed all the way from the last to the first stage. In general, when replacements or deletions are issued on every cycle, each stage must have bypass paths from *all* stages below it; we can avoid such expensive global bypasses by issuing one or more insertions, or by allowing an idle clock cycle between consecutive replacements or deletions.

3.4 Managing a Forest of Multiple Heaps

In a system that employees hierarchical scheduling (section 2), 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 figure 3, 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.

Assume that N heaps must be managed, each of them having a maximum size of 2^{n-1} entries. Then, n stages will be needed; stage j will need a memory L_j of size $N \times 2^{j-1}$. In many cases, the number of flows in individual aggregates may be allowed to grow significantly, while the total number of flows in the system is restricted to a number M much less than $N \times (2^n - 1)$. In these cases, we can economize in the size of the large memories near the leaves, L_n, L_{n-1}, ..., at the expense of additional lookup tables T_j; for each aggregate a, $T_j[a]$ specifies the base address of heap a in memory L_j. For example, say that the maximum number of flows in the system is $M = 2^n$; individual heaps are allowed to grow up to $2^n - 1 = M - 1$ entries each. For a heap to have entries at level L_j, its size must be at least 2^{j-1}; at most $M/2^{j-1} = 2^{n-j+1}$ such heaps may exist simultaneously in the system. Thus, it suffices for memory L_j to have a size of $(2^{n-j+1}) \times (2^{j-1}) = 2^n$, rather than $N \times 2^{j-1}$ in the original, static-allocation case.

3.5 Are Replace Operations needed in High Speed Switch Schedulers?

Let us note at this point that, in a typical application of a heap manager in a high speed switch or router line card, replaceMin operations would not be used –split deleteMin-insert transactions would be used instead. The reason is as follows. A number of clock cycles before transmitting a packet, the minimal entry E_m is read from the heap to determine the flow ID that should be served. Then, the flow's service interval is read from a table, to compute the new service time for the flow. A packet is dequeued from this flow's queue; if the queue remains non-empty, the flow's entry in the heap, E_m, is replaced with the new service time. Thus, the time from reading E_m until replacing it with a new value will usually be several clock cycles. During these clock cycles, the heap can and must service other requests; effectively, flows have to be serviced in an interleaved fashion in order to keep up the high I/O rate. To service other requests, the next minimum entries after E_m have to be read. The most practical method to make all this work is to treat the read-update pair as a split transaction: first read and delete E_m –so that the next minima can be read– then later reinsert the updated entry.

3.6 Comparison to P-Heap

As mentioned earlier, a parallel and independent study of pipelined heap management was made by Bhagwan and Lin [Bhagwan00]. The Bhagwan/Lin paper introduces and uses a variant of the conventional heap, which they call *P-heap*. In this paper, we use a conventional heap. In a conventional heap, all empty entries are clustered in the bottom and right-most parts of the tree; in a P-heap, empty entries are allowed to appear anywhere in the tree, provided all their children are also empty. Bhagwan & Lin argue that a conventional heap cannot be easily pipelined, while their P-heap allows pipelined implementation. In our view, however, the main or only advantage of P-heap relative to a conventional heap, when pipelining is used, is that P-heap avoids the need for the lastEntry bypass, in figure 3; this is a relatively simple and inexpensive bypass, though.

On the other hand, P-heap has two serious disadvantages. First, in a P-heap, the issue rate of insert operations is as low as the issue rate of (consecutive) delete operations, while, in our conventional heap, insert operations can usually be issued *twice* as frequently as (consecutive) deletes (section 4). The reason is that our insert operations know a-priori which path they will follow, while in a P-heap they have to dynamically find their path (like delete operations do in both architectures). Second, our conventional heap allows the forest-of-heaps optimization (section 3.4), which is not possible with P-heaps.

Regarding pipeline structure, it appears that Bhagwan & Lin perform three *dependent* basic operations in each of their clock cycle: first a memory read, then a comparison, and then a memory write. By contrast, this paper adopts the model of contemporary processor pipelines: a clock cycle is so short that only one of these basic operations fits in it. Based on this model, the Bhagwan/Lin issue rate would be one operation every *six* (6) short-clocks, as compared to 1 or 2 short-clocks in this paper. This is the reason why [Bhagwan00] need no pipeline bypasses: each operation completely exits a tree level before the next operation is allowed to enter it.

4 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. This was discussed in sections 3.2 and 3.3, and, as noted, requires 2-port memory blocks that are 4-entry wide, plus expensive global bypasses. 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 5) each heap entry is 32 bits –an 18-bit priority value and a 14-bit flow ID; thus, 128-bit-wide memories are needed in this configuration. 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 discussed in section 3.3 and 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.

Dual-ported memories cost twice as much as single-ported memories of the same capacity and width, hence they are a prime target when trying to reduce cost. Every operation needs to perform one read and one write-back access to every memory level, thus, when using single-port

Pipelined Heap Cost-Performance Tradeoffs							
LINE	COST					$\frac{1}{Performance}$ (CPI)	
	On-Chip SRAM		Bypass Path Complexity	Off-Chip SRAM			
	num. of Ports	Width (entries)		Width (entries)	Levels contained	Cycles per Delete Op	Cycles Per Insert Op
(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
(vii)	1	1	local	2	1	5	2
(viii)	1	1	local	1	1	6	2
(ix)	1	1	local	2	2	5	4
(x)	1	1	local	1	2	6	4

Table 1: Cost-Performance Tradeoffs With Various Memory Configurations & Characteristics.

memories, every operation will cost at least 2 cycles: lines *(iii)* and below. If we still use 4-wide memories, operations can ripple down at 1 level/cycle; given that deletions cannot be issued more frequently than every other cycle, inexpensive local bypasses suffice.

A next candidate to reduce cost is memory width. Silicon area is not too sensitive to memory width, but power consumption is. In the 4-wide configurations, upon deletions, we read 4 entries ahead of time, to discover in the next cycle which 2 of them are needed and which not; the 2 useless reads consume extra energy. If we reduce memory width to 2 entries, delete operations can only ripple down 1 level every 2 cycles, since the aggressive overlapping of figure 5 is no longer feasible. If we still insist on issuing operations every 2 cycles, successive delete operations can appear in successive heap levels at the same time, which requires global (expensive) bypasses (line *(iv)*). What makes more sense, in this lower cost configuration, is to only have local bypasses, in which case delete operations consume 3 cycles each; insert operations are easy, and can still be issued every other cycle (line *(v)*). A variable-length pipeline, with interlocks to avoid write-back structural hazards, is needed. More details on how this issue rate is achieved can be found in [Ioann00, Appendix A]. When lower performance suffices, single-entry-wide memories can be used, reducing the ripple-down rate to 1 level every 3 cycles. With local-only bypasses, deletions cost 4 cycles (line *(vi)*). Insertions can still be issued every 2 cycles, i.e. faster than the ripple-down rate, if we arrange each stage so that it can accept a second insertion before the previous one has rippled down, which is feasible given that memory throughput suffices.

Lines *(vii)* through *(x)* of table 1 refer to the option of placing the last one or two levels of the heap (the largest ones) in off-chip SRAM[8], in order to economize in silicon area. When two levels of the heap are off-chip, they share the *single* off-chip memory. We consider off-chip memories of width 1 or 2 heap entries. We assume zero-bus-turnaround (ZBT)[9] SRAM; these accept clock frequencies up to 166 MHz, so we assume no slow-down of the on-chip clock. For delete operations, issue rate is limited by the following loop delay: read some entries from a heap level, compare them to something, then read some other entries whose address depends on the comparison results. For off-chip SRAM, this loop delay is 2 cycles longer than for on-chip SRAM, hence delete operations are 2 cycles more expensive than in lines *(v)* and *(vi)*. Insert operations have few, non-critical data dependencies, so their issue rate is only restricted by resource (memory) utilization: when a single heap level is off-chip their issue rate stays unaffected; when two heap levels share a single memory (lines *(ix)*, *(x)*), each insertion needs 4 accesses to it, hence the issue rate is halved.

5 Implementation

We have designed a pipelined heap manager as a core integratable into ASIC's, in synthesizable Verilog form. We chose to implement version (ii) of Table 1 with the 2-port, 4-wide memories, where operations ripple down at the rate of one stage per cycle. The issue rate is one operation per clock cycle, except that one or more insertions or one idle (bubble) cycle is needed between successive delete operations in order to avoid global bypasses (section 3.3). Replace operations are not supported, for the reason explained in section 3.5 (but can be added easily). Our design is configurable to any size of priority queue. The central part of the design is one pipeline stage, implementing one tree level; by placing as many stages as needed next to each other, a heap of the desired size can be built. The first three and the last one stages are variations of the generic (central) pipeline stage.

5.1 Datapath

The datapath that implements insert operations is presented in figure 7 and the datapath for delete operations is in figure 8. The real datapath is the mergence of the two. Only a single copy of each memory block exists in the merged datapath, with multiplexors to feed its inputs from the two constituent datapaths. The rest of the two datapaths are relatively independent, so their mergence is almost their sum. Bypass signals from one to the other are labeled *I2, I3,...; D2, D3,...*.

In figure 7, the generic stages are number 3 and below. Stage 1 includes interface and entry count logic, also used for insertion path evaluation. In the core of the generic stage, two values are compared, one flowing from the previous stage, and one read from memory or coming from a bypass path. The smaller value is stored to memory, and the larger one is passed to the next

[8] single-ported, of course

[9] IDT trademark; see e.g. http://www.micron.com/mti/msp/html/zbtds.html

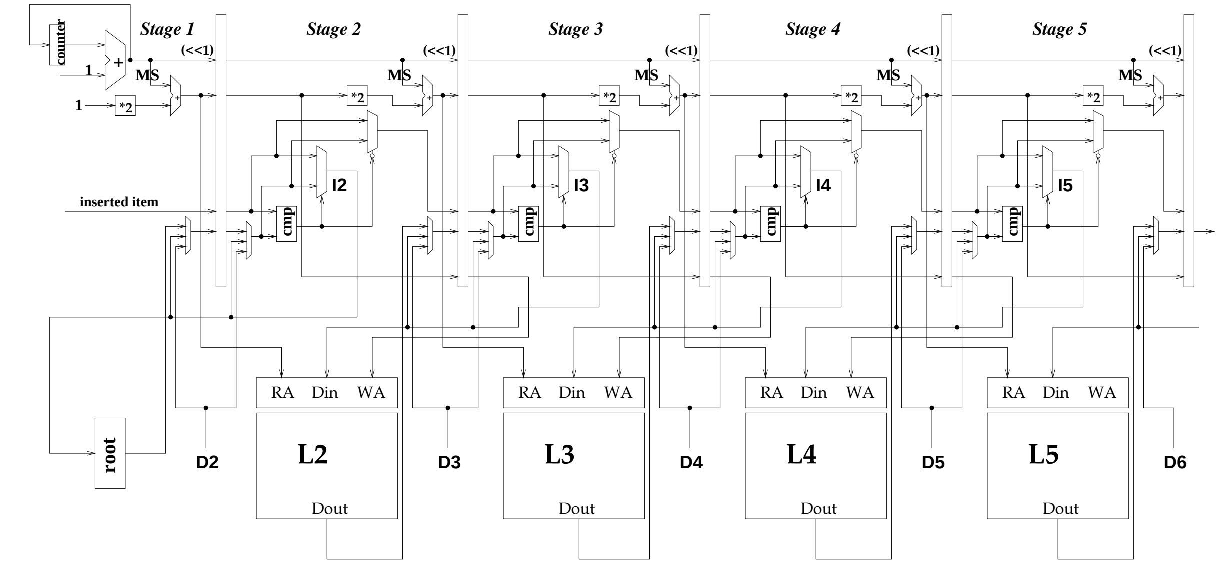

Figure 7: Datapath portion to handle insert operations

stage. Bit manipulation logic (top) calculates the next read address. In figure 8, the generic stages are number 4 and below. The four children that were read from memory in the previous stage feed multiplexors that select two of them or bypassed values; selection is based on the comparison results of the previous stage. The two selected children and their parent (passed from the previous stage) are compared to each other using three comparators. The results affect the write address and data, as well as the next read address.

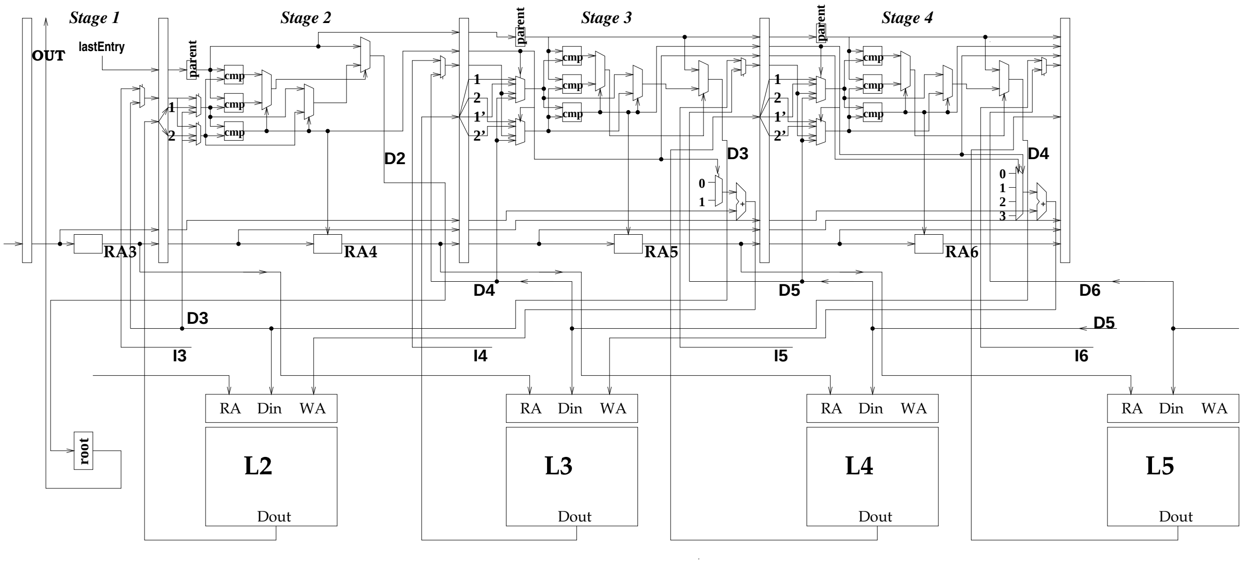

Figure 8: Datapath portion to handle delete operations

5.2 Comparing Priority Values under Wrap-around

Arithmetic comparisons of priorities must be done carefully, because these numbers usually represent time stamps that increase without bound, hence they wrap-around from large back to small values. Assume that the maximum priority value stored in the heap, viewed as an infinite-precision number, never exceeds the stored minimum by more than $2^p - 1$. This will be true if, e.g., the service interval of all flows is less than 2^p, since any inserted number will be less than $m + 2^p$, where m was a deleted minimum, hence no greater than the current minimum. Then, we store priority values as unsigned $(p+1)$-bit numbers. When comparing two such numbers, A and B, if A-B is positive but more than 2^p, it means than B is actually larger than A but has wrapped around from a very large to a small value.

5.3 Verification

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. The top level model, written in Perl, is a priority queue that just verifies that the entry returned upon deletions is the minimum of the entries so far inserted and not yet deleted. The next more detailed model, written in C, is a plain heap; its memory contents must match those of the Verilog design for test patterns that do not activate the insertion abortion mechanism (section 3.1). However, when this latter mechanism is activated, the resulting layout of entries in the pipelined heap may differ from that in a plain heap, because some insertions are aborted before they reach their "equilibrium" level, hence the value that replaces the root on the next deletion may not be the maximum value along the insertion path (as in the plain heap), but another value along that path, as determined by the relative timing of the insert and delete operations. Our most detailed C model precisely describes this behavior. 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.

5.4 Implementation Cost and Performance

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. For the example implementation mentioned above, the memory occupies about 3/4 of the total area. The datapath and control of the general pipeline stage has a complexity of about 5.5K gates[10] plus 500 bits of flip-flops/registers. As we mentioned, the first level is the

[10]simple 2-input NAND/NOR gates

one that varies significantly from the general one, beeing quite simpler. If we consider the first stage together with the extra logic after the last stage, these two together approximately have the complexity of one general stage. Thus we can deduce a simplified formula for the cost of this heap manager as a function of its size (the number of entries it can support):

$$Cost = \ell \times (5.5K\ gates + 500\ register\_bits) + 2^{\ell} \times 32\ memory\_bits$$

where ℓ is the number of levels ($\ell = \log(\#\ entries)$). 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, can be seen in Table 2, where we also see the cost for other heap sizes, separately for memory and datapath. These results are also plotted in figure 9 for ease of understanding. We see that for less than 1K entries the datapath area becomes dominant.

The clock frequency is approximately 180MHz for a heap of 16K entries, in a conservative 0.18-micron technology[11]; the limiting factor that determines this clock frequency is the cycle time of the largest memory block[12]. This clock frequency is for a low-power, low-speed technology. In a higher-speed 0.18-micron CMOS technology, we estimated that the clock frequency could reach 250 to 300MHz. Even, with a clock speed of 200MHz, this heap provides a throughput of 200 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.

Heap Entries	Levels	Mem Area (mm^2)	(%)	dpath area (mm^2)
512	9	2.1	(43)	2.8
1K	10	3.1	(50)	3.1
2K	11	4.3	(56)	3.4
4K	12	6.3	(63)	3.7
8K	13	9.2	(70)	4.0
16K	14	14.5	(77)	4.3
32K	15	23.4	(84)	4.6
64K	16	40.5	(89)	4.9

Table 2: Approximate memory and datapath area in a 0.18-micron CMOS ASIC technology, for varying number of entries

[11] We determined this clock frequency as follows. A 0.18 um library was not available at our site, due to commercial non-disclosure restrictions. Thus, we performed our synthesis using the 0.35 um library of our ATLAS I chip design [Korn97]. Then we extrapolated the 0.35 um area and time/delay numbers to 0.18 um technology, based on extrapolation factors supplied by the manufacturer.

[12] The lastEntry bypass (figure 3) needs a large bus or multiplexer, which cannot be accomodated within a single clock cycle, when the clock cycle is determined by the cycle time of the largest SRAM block. In our detailed design, which was verified as described in section 5.3, the lastEntry bypass was placed in a single clock cycle, resulting in an increase by 60% of the clock period. In a newer design (not in full detail), the lastEntry bypass is broken into two clock cycles (read and local muxing; then long-distance muxing), resulting in the clock frequency of 180MHz.

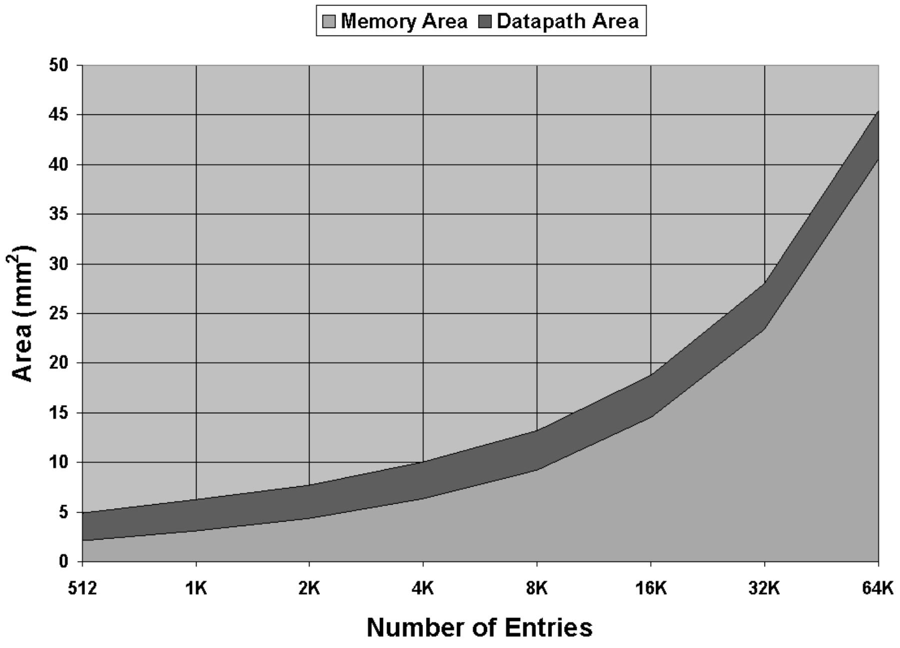

Figure 9: Approximate memory and datapath area in a 0.18-micron CMOS ASIC technology, for varying number of entries

6 Conclusions

We proposed a modified heap management algorithm that is appropriate for pipelining the heap operations, and we designed a pipelined heap manager, thus demonstrating the feasibility of large priority queues, with many thousands of entries, at a reasonable cost, and with throughput rates in the hundreds of million operations per second. The cost of these heap managers is the (unavoidable) SRAM that holds the priority values and the flow ID's, plus a dozen or so pipeline stages of a complexity of about 5.500 gates and 500 flip-flops each. This compares quite favorably to calendar queues –the alternative priority queue implementation– with their increased memory cost and their complexity for handling hash collisions.

The feasibility of priority queues with many thousands of entries in the hundreds Mops range has important implications for advanced QoS architectures in high speed networks. Most of the 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, for many thousand of flows, at OC-192 (10 Gbps) and higher line rates.

Acknowledgements

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

[Ioann00] Aggelos D. 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

[Bennett97] J. Bennett, H. Zhang: "Hierarchical Packet Fair Queueing Algorithms", *IEEE/ACM Trans. on Networking,* vol. 5, no. 5, Oct. 1997, pp. 675-689.

[Bhagwan00] 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

[Brown88] 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.

[Chao91] 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.

[Chao97] H. J. Chao, H. Cheng, Y. Jeng, D. Jeong: "Design of a Generalized Priority Queue Manager for ATM Switches", *IEEE Journal on Sel. Areas in Commun. (JSAC),* vol. 15, no. 5, June 1997, pp. 867-880.

[Chao99] 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.

[Jones86] 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.

[Kate87] 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.

[Kate97] M. Katevenis, lectures on heap management, Fall 1997.

[KaSC91] 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.

[KaSM97] 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

[Keshav97] S. Keshav: "An Engineering Approach to Computer Networking", *Addison Wesley,* 1997, ISBN 0-201-63442-2.

[Niko01] A. Nikologiannis, M. Katevenis: "Efficient Per-Flow Queueing in DRAM at OC-192 Line Rate using Out-of-Order Execution Techniques", *IEEE International Conference on Communications*, Helsinki, June 2001, http://archvlsi.ics.forth.gr/muqpro/queueMgt.html

[Korn97] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis: "Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control", *Proc. 17th Conf. on Advanced Research in VLSI (ARVLSI'97),* Univ. of Michigan at Ann Arbor, MI USA, Sep. 1997, pp. 127-144; http://archvlsi.ics.forth.gr/ atlasI/ atlasI_arvlsi97.ps.gz

[Kumar98] 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.

[Mavro98] 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

[Stephens99] 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

[Zhang95] H. Zhang: "Service Disciplines for Guaranteed Performance in Packet Switching Networks", *Proceedings of the IEEE,* vol. 83, no. 10, Oct. 1995, pp. 1374-1396.