Massively Parallel Priority Queue for High-Speed Switches and Routers

KARI SEPPÄNEN
Information Technology Research Institute
Technical Research Centre of Finland (VTT)
PO Box 1202, FIN-02044 VTT
Finland

Abstract: — The need for better quality of service is growing as new quality sensitive services are becoming more and more important in data networks. The key element in providing quality-of-service or grade-of-service in packet networks is deciding when or in what order the incoming packets should depart the system. In other words the departing packets have to be somehow scheduled. Implementing this scheduling becomes harder and harder as line-speeds and number items to schedule increases. One way to implement the scheduling function is to use a priority queue that can be realised by using simple and scalable heap data structure. The problem is that traditional sequential heap cannot be made fast enough. In this paper a massively parallel system based on set of heaps is presented. It will be shown that such system can be scaled to support very large priority queues at STM-256 speeds and beyond.

Key-Words: — IP routers, packet switches, priority queue, parallel heap, traffic scheduling

1 Introduction

Traditionally data networks have offered only best-effort transport service which have been adequate for applications like file-sharing and e-mail. While lightly loaded local area networks have been successful in supporting some real-time applications, best-effort networks are not enough when these services are to be delivered for large number of subscribers in more complex networks. Thus it is important to guarantee certain level of service for real-time applications to make them usable. This can be done by guaranteeing certain parameters like transfer delay and capacity for each connection or flow (as in ATM and IntServ) or by differentiating the given service based on priority classes (as in DiffServ) [1]. In either case this means that switches and routers cannot use simple first in first out (FIFO) scheduling. On the contrary, more or less complex methods have to be used to ensure that all connections, flows, or service classes get their intented share of service [2].

There are multiple ways to realise traffic scheduling mechanisms varying in complexity and scalability. In simple cases such as a router with few ports supporting only few service classes it is sufficient to use some simple scheduling mechanism based on, e.g. round-robin scheduling. However, even adding just MPLS into this scheme can rise the number of entities to schedule from few to hundreds. With more complicated service systems it is necessary schedule at least 1000s of entities. This implicates that good scalability is required in all but simplest schedulers. One very scalable solution is heap based priority queue that provides $O(\lg n)$ time complexity and linear space scalability[3]. The problem with heap data structure is that each operation takes multiple steps to complete while single-step operations would be preferred in high-speed systems.

In this paper I will propose a hybrid priority queue system that allows for processing one insertion and one extraction at each clock cycle without any conflicts. The proposed system utilises large scale parallelism and its complexity can be tailored to match required operation speed and available system clock speed. It will be shown that the proposed system could support transmission speeds beyond tomorrows technology.

2 Related Work

The way to enhance the operation speed of heap processing is to apply concurrency either by pipelining or parallel processing. There have been some earlier work with parallel heaps, e.g. [4, 5], but the re-

sults are not applicable here: they were dealing with concurrent access to a single heap, i.e. heap mechanisms for multiprocessors. In this paper the target domain is fast access to concurrent heaps. Another way to increase the performance of a heap based priority queue system is to pipeline the heap operations. In the rest of this section is used to introduce the principles and shortcomings two of such proposal.

The P-heap (Pipelined-heap), proposed by Bhagwan and Lin [6], is based on modified (or relaxed) heap structure and top-down, i.e. reverse direction, insertion operation (*ins-op*). In practise, this means that the lowest level ("hem") of the heap can be disjoint, i.e. P-heap resembles more an ordinary binary tree than a proper heap. To find an empty entry for *ins-op* in the P-heap structure a set of so called valid paths is maintained. These paths are used to steer *ins-op* into "left-most" subtree with at least one empty entry. In practise this steering is done by examining the capacity value, i.e. number of free entries, of the left subtree stored in each node[1]. By examining the capacity value *ins-op* can proceed to an adequate subtree. The proposed implementation contains a set of P-heap processors and an on-chip memory system containing a separate memory bank for each heap layer. As different memory banks can be accessed concurrently by different P-heap processors, the execution of heap operations can be pipelined.

P-heap has some disadvantages: the pipeline cannot accept new operations in each operation period and maintaining the capacity information consumes some memory. The reason for scheduling operations only for each other cycle is quite fundamental: the delete operation (*del-op*) manipulates two adjacent levels of the heap and thus the consecutive operations must have one empty cycle between them. As each operation cycle contains three operations (read-compare-write), a new operation can be started in each 6th clock cycle. The book-keeping of the P-heap structure consumes more or less the expensive on-chip memory depending on the actual implementation. If the entries in each level are similar, i.e. the width of the capacity value is constant, the memory consumption can be quite high equalling l-bits per entry for l-level P-heap. This means that, e.g. in 2^{16}-entry P-heap with 32-bit priority values 1/3 of the memory is consumed by capacity values[2]. Fortunately, considerable amounts of memory can be saved if variable length values are allowed — it turns out that only about two bits per entry is required on the average. There is still one minor oddity in P-heap algorithms: *ins-op* proceeds always to left subtree if possible. This means that if there is $\geq l$-entries in 2^l-entry P-heap it is very probable that *ins-op* has to travel through all levels. While it does not affect into the computational performance of the P-heap, it will most certainly increase power consumption and thus heat generation. However, this can be easily cured by steering the insertions towards the empties subtree.

Another pipelined heap, proposed by Ioannou and Katevenis [7], is based on true binary heap with modified insertion mechanism. The *ins-op* was modified to operate from top to bottom (as with the P-heap), i.e. in the same direction as *del-op*, to allow for pipelining all operations. It is proposed that there could be one (or even multiple) operation in process at each level of the heap (there cannot be two consecutive *del-ops*). To enable pipelining their proposal is using similar on-chip memory system as above. However, the modified insert operation described in [8] works correctly only if it is known how many delete operations there will be until the insert operation has completed. The insert operation relies on steering the insertion into the correct position. If the correct position is not know in advance, the heap will corrupt, and in fact, become very similar to P-heap. However, the operations developed for this heap work only with proper heaps.

I think that in a system using variable length packets such as an IP router it is quite impossible to predict the number of *del-ops*. One can imagine a situation where a entry is inserted into a heap that has full left sub-heap and right sub-heap has only one entry at the lowest level. Furthermore, the top of the heap points to a packet that has length of 1500 octets which should mean that there will be only one *del-op* before the *ins-op* in-process completes. So in this situation it is correct to *steer* the *ins-op* to the right sub-heap. However, if a subsequent *ins-*

[1] It seems like the authors have forgotten or misplaced sufficient capacity value updates in their `local-enqueue` algorithm shown in the Figure 4 in their paper.

[2] Somehow the authors explain the capacity requirements in a way that can be easily misunderstood: one can get an impression that 256 KB memory can host up to 2^{17} 32-bit entries. However, such 2^{17}-entry heap with 4-byte priority values uses twice the memory excluding the expenses of book-keeping.

op replaces the top entry with, e.g. 40-octet packet, and causes **two** *del-ops* instead of one, the steering decision made earlier would be now incorrect. Similar troubles will occur of course in a reverse case if two *del-ops* are predicted and only one occurs. To avoid corruption of the heap structure there are two options; either to force the predicted amount of *del-ops* to occur or interrupt the *ins-op* heading towards an incorrect position and re-issue it from the beginning. Another option is to modify the *del-op* algorithm: instead of terminating youngest-in-process *ins-op* (and using its value to replace the top of the heap) as proposed, the oldest-in-process or mis-steered one should terminated. However, this modification has some side-effects: *del-op* and *ins-op* cannot be any more interleaved as proposed but there must be one empty cycle before each *del-op*[3] In conclusion it can be said that this version of pipelined heap would require quite complex management procedures to maintain the heap uncorrupted if the proposed performance level[4] is wished to be achieved and thus it is not so attractive alternative.

It seems that it is quite complicated to achieve extremely high performance priority queue mechanism by pipelining the heap operations. The speed of each pipeline stage is limited by the timing of read-compare-write cycle and as *del-op* depends on two adjacent levels it is hard or maybe impossible to start new operations at each operation cycle. It is clear that if a higher performance and simpler alternative is sought, new approaches are required. One way to find a new solution is to use multiple parallel heaps together with a system that can be used to maintain the top entries of the heaps sorted. In this way each heap can be processed in ordinary sequential form without any needs to modify heap algorithms or structure. Moreover, the performance is not limited by the implementation of heap operation stages but can be scaled up by increasing the number of concurrent heaps. The limiting subsystem in this scheme will be sorting array but, as its size is limited, it can be easily made to operate at speed of one insertion and deletion per clock cycle.

[3] Actually in no phase there is any operation in-process at a stage below advancing *del-op*. Each *del-op* begins with terminating the youngest-in-process *ins-op* and thus the stage before *del-op* is *always* inactive. This is not so surprising as this heap implementation is bound by same limitations as P-heap.

[4] In their paper Ioannou and Katvenis claim that their system can achieve 200 million operations per second (Mops) at 200 MHz and thus the system throughput would be 64 Gbit/s with 40-octet packets. First of all, no information was provided how the operations are scheduled if they are allowed to overlap (starting new operation at each clock cycle). Based on the properties of the heap structure I doubt that such scheduling can be created. Thus it is likely that the performance of their system is limited to 66 Mops at 200 MHz. Furthermore, the processing of each packet requires **two** operations (*ins-op* and *del-op*) reducing the performance to 33 Mpackets/s. Finally, if the *del-op* have to be modified and thus requires idle operation to be issued before *del-op*, the performance goes down to 22 Mpackets/s. This means that the worst case performance of their system could be only mere 7 Gbit/s at 200 MHz.

3 Massively Parallel Priority Queue

First of all we should note few facts with the scheduling of packets departures in high-speed networks that can be used to lessen the requirements. The most important fact is that absolutely exact ordering is not required, i.e. it is not a failure if a packet gets served one or two packets later (one 1500 octet packet "lasts" 1.2 μs at STM-64 rates). This means that we can have some latency in the processing. However, it will be shown that the proposed Massively Parallel Priority Queue (MPPQ) system can process insertions with sub-microsecond latencies. Thus the priority queue will be in order almost always. Another feature that can be utilised is that there is usually a class of traffic without any service guarantees. This best-effort class can be very well served when there is not other traffic and thus does not need to be inserted into the priority queue. By separating the best-effort traffic from higher class one the capacity of the priority queue can be reduced considerably.

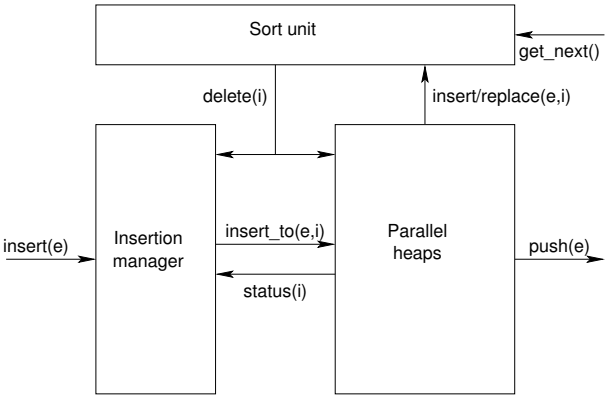

Figure 1: The architecture of massively parallel priority queue.

The MPPQ is composed of three major blocks shown in Figure 1. The Insertion Manager (IM) selects a suitable heap for new insert operation depending on the status of all heaps and optionally depending on ongoing extract operation. The Parallel Heap Unit (PHU) contains the actual heap structure and heap managers. The Sort Unit (SU) will get the top most entries form all heaps and sorts them. When get_next() operation is issued to the system SU will request the PHU to extract the maximum value.

3.1 Parallel Heap Unit

PHU contains (Figure 2) a heap unit controller (HUC) and a set of heaps. HUC is not much more than an intelligent multiplexer that forwards the incoming requests to internal busses and activates correct heap. Furthermore, it forwards the status information from heaps to IM.

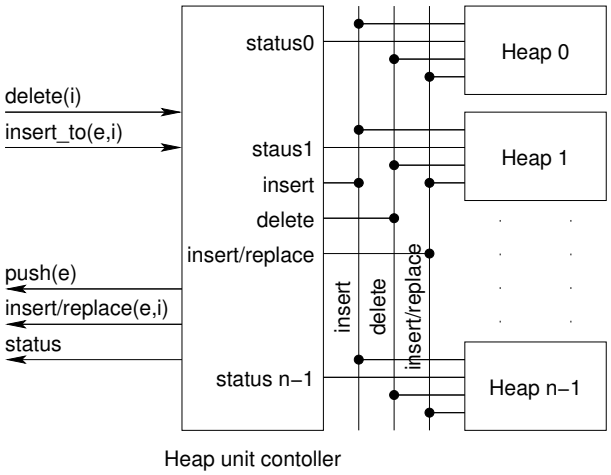

Figure 2: The parallel heap unit

Each heap contains a heap manager (HM) and local SRAM block (Figure 3). HM implements the required heap operations, i.e. insert and extract, while the SRAM block is used to store the heap data structure. HM processes all heap operations in sequential manner using standard heap algorithms [3]. In this way the implementation of HM can be kept as simple as possible saving chip area and allowing for higher system clock speeds. The down side of this simplicity is that each operation takes $3l$ clock cycles with l-level heap at the worst case. So to allow two simultaneous operations for the proposed system, $6l$ concurrent heaps are required. On the other hand, the more heaps there are the lesser l is required for certain queue capacity, e.g. with 10-level heaps at-least 60 heaps are required giving a capacity of 61 Kentries. One way to optimise the performance is to divide each heap into few independent sub-heaps and to add small local sort array into each HM. In this way heap operations will complete earlier reducing the number of required heaps. However, it is not certain that this modification would improve the overall system efficiency.

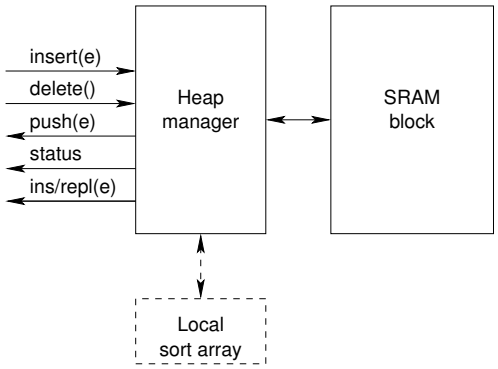

Figure 3: A single heap consisting heap manager and SRAM block. A local sort array is required if there is multiple heaps.

The requirement of single cycle operations for the whole system causes some additions to standard heap algorithms. As *del-op* may be issued to a heap that is processing *ins-op* or *del-op*, a delayed *del-op* has to be introduced. The delayed *del-op* simply allows the extraction of the top most value during an on-going operation (as long as that operation is not accessing the top-most entry). When the on-going operation finalises HM will continue with the pending *del-op*. Of course if *ins-op* proceeds to the top of the heap, it has to be checked if the top-most value is valid any more. A delayed *del-op* can be combined with a new *ins-op* to a delayed insertion-deletion. In such case IM issues new insertion to a heap with pending *del-op*. In combined insertion-deletion the new entry is used to replace the top-of-the-heap and after that *del-op* proceeds as normal (this is quite similar to enqueue-dequeue operation in [6]).

To allow for sorting of the top-most entries of different heaps each heap manager issues either insert or replace operation for SU. If the heap has been empty before insertion of a new value, HM should issue insert operation. If the heap was non-empty, it already have an entry in SU. In that case

a replace request is issued. Both of these two requests should be issued only after *ins-op* has been completed. With other operations, *del-op* and delayed *del-op-ins-op*, the entry for this heap has been deleted from SU by a get_next() operation and thus insert operation should be issued for SU. This can be done as soon as the top of the stack is valid, i.e. the operation has bypassed the top of the heap.

3.2 Sorter Unit

SU shown in Figure 4 has one sort-stage (SS) for each heap. A SS holds one entry that is composed of priority value and a heap index (HI) of the heap that stores that value. It is capable of comparing the stored entry with an entry flowing down from SS above it. The upper-most SS (sort-stage 0) hold the largest value and upon a get_next() operation requests PHU to extract that value. At the same time all entries in SU are moved one SS upwards. An insert operation causes SS compare the priority value of the new entry with the value stored in the stage or a value pushed up from the stage below. The lager value is stored into the stage and the smaller one pushed downwards. An empty stage accepts any new entry. A replace operation is similar to the insert but an additional discard information is flooded downwards. If HI carried by discard information matches, SS discards the old value. In this way it is make sure that each heap has only one entry in the SU.

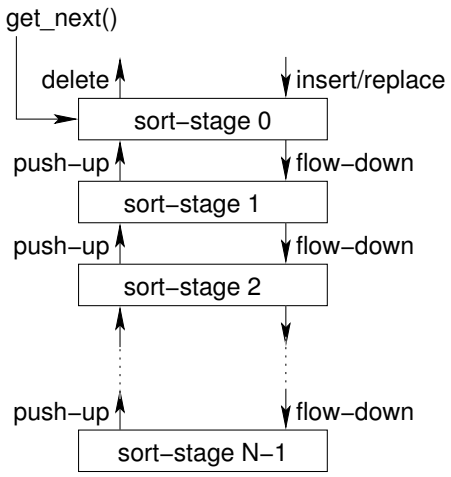

Figure 4: The sorter unit

SU is likely to be the most critical subsystem for timing and thus some means to speedup its operations might be needed. Fortunately, strictly exact ordering is no required but almost exact order is enough. Thus the operations of SS can be pipelined in quite straight-forward manner. In worst case simple pipelining would cause a delay comparable to the number of pipeline stages. However, this should not have any meaningful impact on QoS.

3.3 Insertion Manager

The main task of IM is to keep the depths of heaps close to each other. In this way the average of the number of steps needed for heap operations is kept close to minimum. While the system should be designed to cope with worst case loads, the average power consumption is reduced by minimising the active periods of each heap manager. IM uses the status information provided by each HM through HUC. In the best situation the status information includes exact information of heap occupancy but it might be possible that providing such information is too expensive. Thus the status information could be limited to heap full/empty with optional almost full/empty information. However, this should be enough for IM to keep the variation of heap depths within certain limits.

Another task for the IM is to combine new insertions with pending *del-ops*. This is done by selecting a heap with status indicating pending *del-op* for next insertion operation. If there is multiple pending *del-ops* the heap with oldest status or the emptiest heap should be selected.

3.4 Performance Estimates

At this phase only the architecture and algorithms of the proposed system have been designed while all modelling with VHDL remains yet to be done. So instead of calculating some performance estimates based on some hypothetical clock speeds, minimum clock speeds can be estimated based on the performance requirements. When STM payload sizes and minimum IP packet length of 40-octets are used the required clock speeds are about 30 MHz for STM-64, 120 MHz for STM-256, and 480 MHz for STM-1024. This shows that speeds of 10-40 Gbit/s could be supported even by current FPGA technology and with STM-64 speeds the degree of parallelism can be quite well reduced. Furthermore, even 160 Gbit/s speed could be supported by today's VLSI technology.

The maximum processing delay can be calculated directly from the number of levels in heaps, e.g. a system with 10-level heaps at 200 MHz has maximum delay of 150 ns. The combining of delayed *del-op* – new *ins-op* has no impact on this, as the new value is inserted into the top of the heap. Thus the worst case wait time for delayed *del-op* is more or less compensated by faster validation of the top of the heap. This means that "out-of-orderness" should be well below 1 μs in realistic system configurations and thus have negligible impact on QoS.

4 Conclusions

The proposed Massively Parallel Priority Queue has been shown to be capable not only to cope with latest SDH/SONET 40 Gbit/s line-speeds but to be scalable for over 100 Gbit/s speeds. This formidable level of performance has been achieved by using a large number of parallel heaps to overcome the fundamental limitations on speeding up operations on single heap. Thus MPPQ is capable of performing operations at the rate of two per clock cycle which is clear improvement compared to pipelined systems that are limited into one operation in each 3rd-6th cycle. Furthermore, the queue capacity of a MPPQ based system can be easily tailored by matching the number of concurrent heaps with the depth of the heaps. As MPPQ does not require any excess bookkeeping the expensive on-chips memory is used as efficiently as possible.

These results with MPPQ can give an inspiration to think if not only STM-256 capable IP and ATM line cards are possible but the speeds well beyond that. The trick could be in tweaking all the subsystems to operate at a rate of single cycle per an operation. It seems that it could be quite possible with, e.g. address lookup [9]. However, there are many subsystems like buffer management and packet classification that would require further study to find out if they can be made to operate even at the rates required by 40 Gbit/s interfaces.

References

[1] V.P. Kumar, T.V. Lakshman, and D. Stiliadis. Beyond best effort: Router architectures for the differentiated services of tomorrow's internet. *IEEE Communications Magazine*, 36(5):152–164, May 1998.

[2] Hui Zhang. Service disciplines for guaranteed performance service in packet-switching networks. *Proceedings of the IEEE*, 83(10):1374–1396, October 1995.

[3] Thomas H. Cormen, Charles E. Levison, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[4] Sushil K. Prasad and Sagar I. Sawant. Parallel heap: A practical priority queue for fine-to-medium-grained applications on small multiprocessors. In *Proceedings of The 7th Symposium on Parallel and Distributed Processing*, 1995.

[5] V. Nageshware Rao and Vipin Kumar. Concurrent access of priority queues. *IEEE Transactions on Computers*, 37(12):1657–1665, December 1988.

[6] Ranjita Bhagwan and Bill Lin. Fast and scalable priority queue architecture for high-speed networks. In *Proceedings of IEEE INFOCOM 2000*, pages 538–547, 2000.

[7] Aggelos Ioannou and Manolis Katevenis. Pipelined heap (priority queue) management for advanced scheduling in high-speed networks. In *Proceedings of IEEE ICC 2001*, 2001.

[8] Ioannis Mavroidis. Heap management in hardware. Technical Report TR-222, FORTH-ICS, July 1998.

[9] Kari Seppänen. Novel IP address lookup algorithm for inexpensive hardware implementation. Accepted to 6th WSEAS ICCOM'02.