FORTH-ICS / TR - 278 November 2000

An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks

Aggelos D. Ioannou

ABSTRACT

Quality-of-Service (QoS) guarantees in networks will soon be provided using per-flow queueing and sophisticated schedulers. 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 analyze the cost-performance tradeoffs: using 2-port, 4-wide SRAM's and global bypasses, heap operations can be initiated at the rate of one per clock cycle; when reducing these costs, issue rates drop to one every few cycles. Our design can be configured to any heap size, and supports initiating operations on every clock cycle, except that one idle (bubble) cycle is needed between two successive delete operations. We have verified our design by co-simulating it with three heap models of varying abstraction. We have also performed synthesis, and are presenting cost analysis information.

An ASIC Core for Pipelined Heap Management to Support Scheduling in High Speed Networks

Aggelos D. Ioannou

Computer Architecture and VLSI Systems Laboratory (CARV)
Institute of Computer Science (ICS)
Foundation for Research & Technology - Hellas (FORTH)
P.O. Box 1385, Science and Technology Park of Crete,
Heraklion, Crete, GR-711-10 GREECE
Tel: +30 81 391660, Fax: +30 81 391661
email: ioannou@ics.forth.gr
http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

Technical Report FORTH-ICS/TR-278 October 2000

Work performed as a M.Sc. Thesis at the Univ. of Crete

ABSTRACT

Quality-of-Service (QoS) guarantees in networks will soon be provided using per-flow queueing and sophisticated schedulers. 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 analyze the cost-performance tradeoffs: using 2-port, 4-wide SRAM's and global bypasses, heap operations can be initiated at the rate of one

per clock cycle; when reducing these costs, issue rates drop to one every few cycles. Our design can be configured to any heap size, and supports initiating operations on every clock cycle, except that one idle (bubble) cycle is needed between two successive delete operations. We have verified our design by co-simulating it with three heap models of varying abstraction. We have also performed synthesis, and are presenting cost analysis information.

This text is available through the web:
http://archvlsi.ics.forth.gr/muqpro/heapMgt.html

Acknowledgments

I would like to thank all those who helped me throughout my work. First of all I would like to thank prof. Manolis Katevenis who gave the idea for the whole work. Also I would like to thank him both for his priceless suggestions and ideas during my work, as for trying to reveal and communicate to me the way one should think while designing and in general while working on scientific research.

I would like to greatly thank George Kornaros who helped with the hardware design tools of CADENCE, both for working on the synthesis of the heap-manager design, and for teaching the tools to me.

Also give my thanks to prof. D. Pnevmatikatos for his helpful remarks on writing (efficiently !) synthesizable Verilog code. Professors A. Traganitis and E. Markatos gave me useful comments on this report.

I would like to thank the Institute of Computer Science of FORTH for providing the fellowship that supported this work, 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.

Finally I thank my family for the help and support throughout my studies.

Contents

List of Figures

List of Tables

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 work 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. 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 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). Increasing sophistication will bring finer flow granularity, and then many thousands of queues will be needed. Maintaining many queues and managing them at high speed incurs a non negligible cost, but is nevertheless feasible using modern semiconductor technology [Korn97].

This work deals with the next problem: implementing sophisticated scheduling algorithms at high speed, when there are many thousands of contending flows. 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, for OC-192 (10 Gbps) and higher rates, and for packets as short as about 50 bytes, 50 Mops or more 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 was proposed by Katevenis in 1997 [Kate97] (see the Technical Report [Mavro98]). In this work we are concerned with a pipelined heap manager that we have designed in the form of a core, integratable into ASIC's.

Section 3 presents an overall aspect of how the structure we are engaged with is operating, after giving an indicative description of its building blocks. This description aims to help us with the discussion that follows.

In section 4 the operations to be performed are reviewed, so that our manager offers the required functionality. The pipeline structure is then explained and a first problem appearing with delete's is presented. Ways to achieve a high execution speed are given and, later on, the overall aspect of the structure (initially presented in section 3) is now refined to better deal with the issues arising with pipelining.

Section 5 gives some alternative algorithms for high pipelined speed, that eventually reach up to the most efficient scheduling, presented in sections 5.4 and 5.5. Achieving the highest operation throughput of 1 op/cc leads to much added complexity (section 5.7) and is thus not adopted, while more alternatives that trade off performance to achieve a lower cost are presented in section 6 .

Finally, section 7 describes our implementation, which is in synthesizable Verilog form. The ASIC core that we have designed is configurable to any size of priority queue. The datapath is presented in detail, along with many problems arising when some techniques are put into hardware. The solutions adopted for each are described. All the bypasses needed for resolving possible data hazards are reviewed in section 7.7 and the memory modules used are described in section 7.8 . The verification process is then described and finally, the resulting cost & performance of the heap manager.

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 work 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.

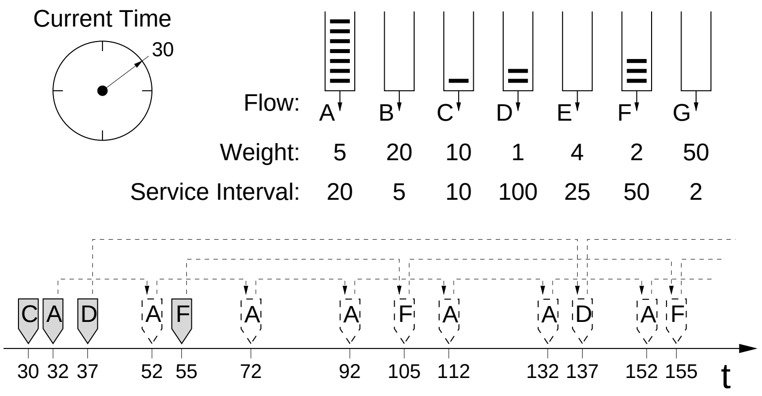

Figure 1: Weighted round robin scheduling

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 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 update (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 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) element in 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 (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

[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.

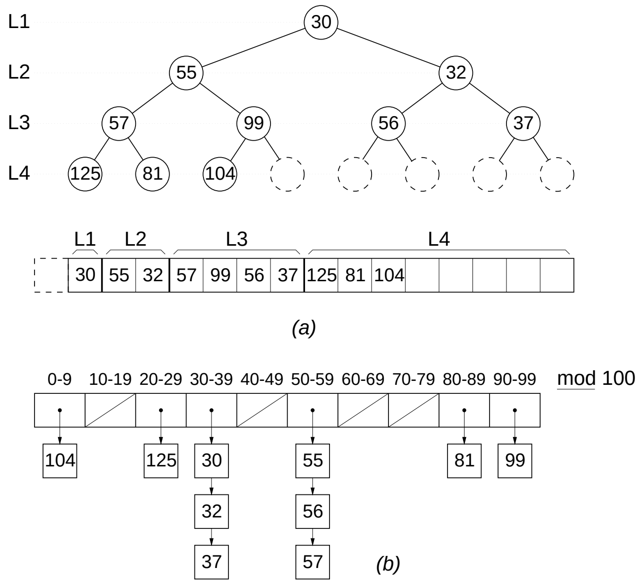

Figure 2: Priority queues: (a) heap; (b) calendar queue

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]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.

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]. Our work 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; 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 work have been reported in the literature, so we cannot make a direct comparison; an indirect comparison follows. 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],

[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-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*.

the number of buckets in 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, the performance that can be achieved using a hardware FSM manager with the heap stored in one or two off-chip SRAM's was studied previously and the results were reported in Technical Report [Mavro98]. In this work, we look at higher-speed heap management, using pipelining.

3 Our Structure and its Invariant Condition

To support a scheduling discipline, we would like to have a structure that stores its elements in a way that information can be extracted according to a priority field. To be more exact, we would like to be able to get the element with the smallest priority in the minimum possible time, O(1) that is, i.e. independent of the number of elements (N) stored in the structure. This does not compel for total ordering of the elements. However, other operations to keep the structure updated should be performed, such as deletions and insertions, and this manipulation of the structure should be carried out efficiently. To achieve this, a partial ordering of the elements can be of great help. So we would like our structure able to store elements, each associated with a numeric value, and giving the element with the smallest value in O(1) time. All the needed updating operations that manipulate the structure should perform similarly, at least to the interface level.

The structure we use to fullfill our objectives is a binary tree which holds its data in such a way that a priority queue is implemented. A tree implementing a priority queue is a tree with each node being less than or equal[5] to its children nodes (already mentioned at section 2.2), also called a heap data structure. Thus the root is the smallest element of all contents of the tree. Furthermore, when we consider efficiency in manipulating such a structure, we want it to have one more property: being balanced and even more, fully balanced, that is having all the levels of the tree being completely filled with only the possible exception of the lowest level which may have its leftmost leaves populated and the rightmost (if any) being free. In this way the distance of the leaves from the root vary at most one and no "holes" appear in the tree. So, we can state that the heap data structure (or just "heap") supports the *Invariant Condition* of being a fully balanced, smallest priority queue, binary tree. The partial ordering of the elements offered by the heap aids for the execution efficiency of the updates.

Now let us see what an element should be. From the discussion so far we know that an

[5] symmetrically, the heap may be defined with each node being greater than or equal to its children

element should consist of two numbers: an *id* and the associated *priority*. The id can be used in many ways, e.g. as a VC number for an ATM network. The priority is the value that determines the relative position of the element in the tree. It is therefore the metric of ordering and comparison among the elements of the heap. In the previous example it could represent the priority of the corresponding VC. Here we would like to mention that in the following discussion and descriptions when we refer to an element being greater or smaller in comparison to another one, we mean to say the corresponding priority number. The same goes for every numeric reference to an element unless explicitly stated otherwise.

Moving even further, we would like to give a more precise aspect on how the state of the tree changes while various operations are evolving, so as to present a helpful viewpoint for the execution of operations.

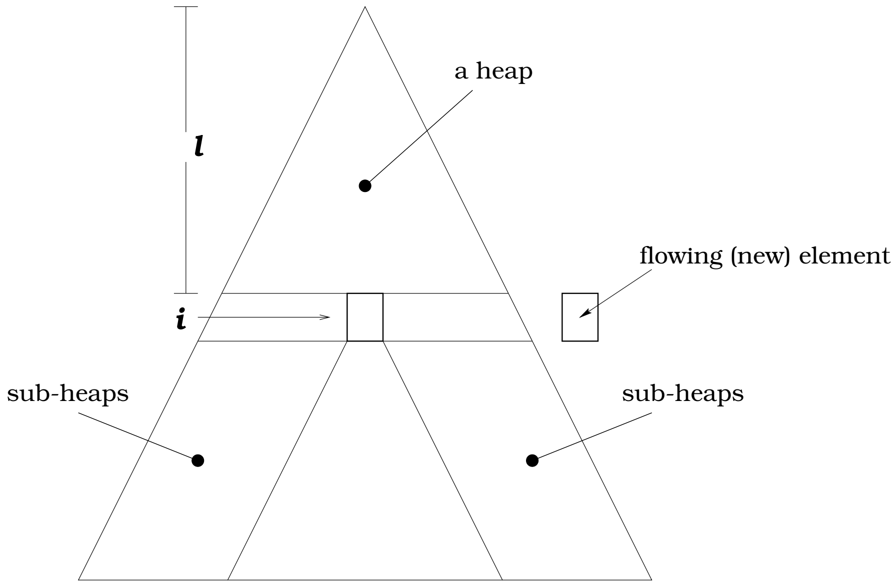

Figure 3: A generic snapshot of the heap state while an instruction is evolving

Figure 3 will help our discussion. Firstly, each operation has an added or otherwise a new element associated with it. For an insert operation the newly inserted item is the one that is new for the structure. For the delete operation an old element is extracted from a leaf and reinserted at the root. This element to be "placed" (temporarily) as the tree root is considered to be the new one. One other thing associated with each operation is the tree level at which it is currently executing (from here on referred to as the "execution level"). For example the operation in figure 3 has traversed **l** levels of the tree and is now executing at level $\mathbf{l+1}$.

The new element is initially next to the root of the heap ($l = 0$). Then, as we move downwards, at each level transition a new element is decided to be the one that will flow besides the next level. This flowing element, chosen after a set of comparisons, will be refered to as the flowing element. Another basic aspect of each operation is the path to be followed from each level on to the next one and entirely from the root down to the leaves of the tree (where each operation completes). For the insert operation this is known from the start; for a delete it is determined while the operation traverses the tree, depending on the outcome of the comparisons executed.

In the figure, **i** is the position of the tree level for which the element to be placed is currently under consideration. So, a position at the execution level is also associated with an operation. This position determines a sub-heap having that element as the root. This sub-heap's root element is the only one that possibly violates the heap property. To be more exact, for a delete operation the flowing element can be considered as the one occupying the root of the sub-heap. (However it is not stored there yet for efficiency purposes). In the case of a delete, this root will be compared with the children of position i to decide the new element to be placed there. For an insert operation the root is occupied and does not violate the property. However the flowing element that is to be inserted, will possibly get exchanged with the sub-heap's root. Nevertheless, in the case of insert, removing the flowing element (e.g. for the purposes of a delete) will leave the heap in an consistent state. The element at position **i** will be compared to the flowing one to decide whether they will be exchanged or not.

4 The Operations Performed On a Tree Structure and Some Problems Encountered

Here we will present the operations that could be supported by a heap manager. We start with a basic presentation of these operations stating their main reason of existence. Then we give a generic description of how these actually operate on the tree structure. Finally we consider performance issues and introduce pipelining. A special problem emerging with pipelining for the delete operation is presented along with its solution.

4.1 The Basic Idea of the Operations Involved

In this section we discuss the operations needed to be performed on a tree implementing a priority queue, so that each possible update or query for the contents of the tree can be executed, and furthermore be fast. The basic operations for our priority queue tree structure, as for every dynamically changing data structure, are *delete* and *insert*. As our tree holds priority information with only the root having a globally distinct property (i.e. being the smallest of all elements) the delete operation has meaning only for the root element. So when we refer to delete we mean delete the root of the tree. Insert is the function that adds an element to the tree. Insert adds this element (at the point of the root) and performs the repositionings need to take place in order to keep the tree's priority queue property intact. Keep in mind that insert conceptually adds a new leaf to the tree, however we will show later why we start from the root. Delete on the other side deletes the root of the tree, and then performs the needed repositionings to place a new root and maintain a balanced tree with no "holes" and with its priority queue property intact. The root element deleted is given as a return value, speaking in terms of a C like programming language. Another operation often needed on tree structures with the attributes of ours is the *replace* operation. This operation is performed to the root element of the tree and adds to its value a signed number given as a parameter. This operation can be synthesized out of our two basic operations (delete and insert), by deleting the root, updating its value as needed and then reinserting the updated element.

This was one of the two reasons replace was not implemented in our design. The second and most important reason is based on the following observation. If we want to update a VC's priority number and particularly the one with the lowest priority number residing at the root, we should firstly be able to know which this VC is. This situation arises for example when we want to service a VC which still has packets to send and should thus remain in the heap, but with a new higher priority (i.e. lower priority number). However there is no way to know the VC number of the root, unless we "dequeue" it through a delete operation. And the VC number has to be known as the value to be added to its current priority depends on the VC's service class. One alternative would be to always drive the root value to an output so that the involved operating module could have it. But in that case, the aforementioned module would need to know the validity periods of the root, depending on the previously issued operations. So it would need to be based on internal implementation details, something that would make the interface much more complicated and dependent on implementation modifications. What is more, in the interval between the delete and the insert operation, 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. Nevertheless, as we will see later when we present the implementation and the corresponding datapaths, the implementation is such that replace can be added very easily because it is in fact a special case of a delete. Only minor and intuitively easy additions need to be carried out.

4.2 A Generic Description of Each Operation

Now we are about to see how these operations actually take place in the tree structure, with all the needed repositionings described.

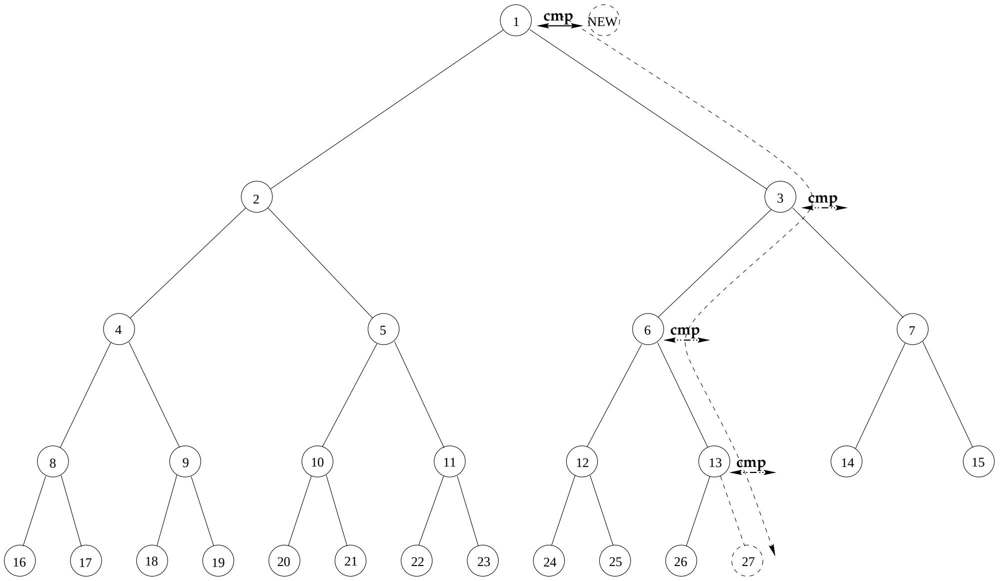

Figure 4: A schematic description of the insert operation

4.2.1 Manipulating the Insert's Top-to-Bottom

Let us start our discussion with the insert operation. The insert operation actually takes as a parameter a new element and has to insert it in the tree. Insert conceptually performs the insertion at the leftmost empty leaf on the bottommost level of the tree, and then rearranges the elements above it so as to keep the priority queue property in effect. However in order to support pipelined heap management, and as proposed in [Kate97], we would like to have all operations proceeding top-to-bottom and so we start the insertion at the root of the tree. The way an insert should operate is presented in [Kate97] and is as follows. Actually insert does not perform the insertion at once. The newly inserted element has to find its correct position into the tree. Correct in the context of keeping the invariant condition of our structure intact. So a new entry must eventually be placed as the left-most empty leaf of the tree (not necessarily containing the element just inserted) thus keeping the tree balanced, and with each node continuing to be smaller than its children nodes. Therefore, insert compares the newly inserted element with the root element and places the smaller as the new root (possibly the same element continues to be the root). It continues to do so recursively as it traverses the tree from top to bottom. The path to be followed is known, since the left-most empty leaf of the tree can be located due to our continuous knowledge of the element count. Finding the route to be followed is therefore intuitively easy, however the implementation is somehow tricky and is presented later on.

4.2.2 Delete's & Replace's

Let us now consider delete operation. Delete operation has to erase the root element (returning its contents as a result). As the root now becomes empty, another element of

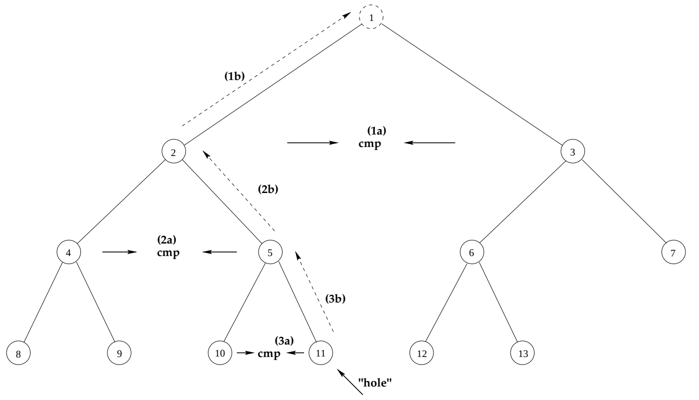

Figure 5: A delete operation leaving a "hole"

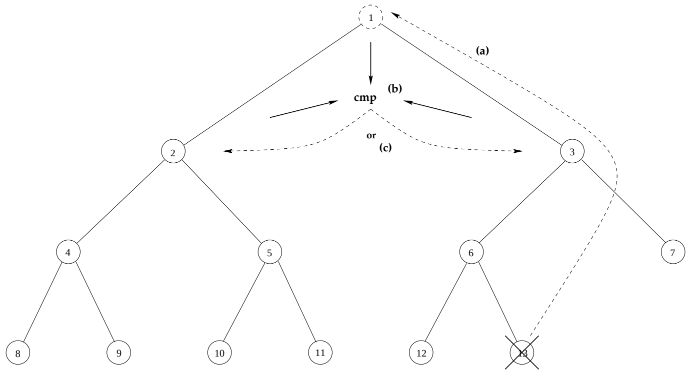

Figure 6: A delete operation avoiding the "hole" creation

the tree should be placed as the new root. If we choose one of its children nodes (the smaller) then a hole would move towards the leaves of the tree, choosing its path "arbitrarily" as it passes through the tree nodes. Here arbitrarily is not to be taken literally. What we mean is that the new "turn" is chosen with the metric of ordering and the path needed to keep the tree balanced is not followed. So this method is not appropriate, as the tree can end up highly unbalanced, leading to difficulties in the pipelined execution of the issued operations. What we would really like is having the new "hole" "placed" in the rightmost leaf currently in use. This can be accomplished by having the latter removed and being placed in the place of the root, which has just become empty. Then the newly inserted root is compared with its children and the smaller of the three gets exchanged with the new root. This is done recursively for the lastly repositioned element, until no repositioning is needed. At that time the delete operation is considered to be finished and the tree remains fully balanced and with its priority queue property in effect.

Finally we discuss the replace operation. As mentioned earlier, this operation was not implemented in our design but it can be easily added, because its implementation resembles the way we implemented the delete operation. The replace operation needs to take a value as a parameter (positive or negative) and adds this value to the root element (i.e. increments root by 'value'). From there on, this operation acts just like delete, after the latter has moved a leaf to the root. A new element is placed as the new root and some elements of the tree have to be rearranged by recursively comparing a parent node with its children in order to maintain the priority queue property in effect.

4.3 Introducing Pipelining

To achieve a high instruction throughput for our structure and the accompanying design we should allow new operations to issue and start executing (i.e. rearranging the elements of the tree) before a previous one has completed. In that way, many operations can be in progress, operating on the elements of the tree. To achieve such parallelism, each operation already issued should manipulate a distinct set of elements. The best way to accomplish this is to have all operations traverse the tree from top to bottom as time elapses, each following the previously issued ones in a pipelined fashion. This is the reason we tried to have all the operations progressing from top to bottom throughout our previous discussion. However, although this way may seem greatly efficient, much added effort is needed for the design, as data hazards are introduced, and forwarding paths and control logic need to be added. Furthermore, issuing each operation right after its previous one (i.e. on the next clock cycle) is very difficult (requires much added hardware) and so is resolving some related data hazards. These are discussed extensively in the following sections.

figure 7 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 two basic operations are provided (insert and delete), while the third operation (replace) can be easily added.

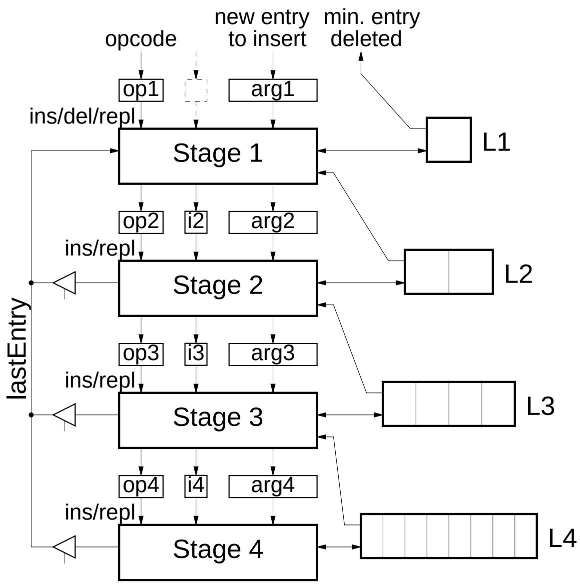

Figure 7: Simplified block diagram of the pipeline

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. Of course, the traditional *insert* algorithm needs to be modified, as presented in section 4.2.1.

4.3.1 Interaction Between Delete's and Topmost Insert

In this section we begin our discussion of the difficulties involved while trying to achieve high efficiency by stating one of the main problems appearing for the delete operation when pipelining is introduced. Its solution is also presented afterwards. When we described delete functions in section 4.2, we said that the rightmost leaf currently in use should be removed. But with pipelining this element is not explicitly known, because some other elements may be flowing on the tree, as other operations traverse it towards the leaves. So when a delete operation is issued, the element to be placed as the rightmost leaf may still have not taken its final position as it could have been issued through a yet incomplete insert operation. If we used the rightmost currently occupied leaf, then the tree would end up containing "holes", as the newly inserted elements continuing to traverse the tree have chosen the position they will be placed at operation issue time. Furthermore it may already be too late changing this decision as a wrong "turn" for the flowing element may have already been taken. So, the element just removed will leave a hole in its position. What we can do is use (remove) the element that would end up to be the rightmost occupied leaf after all the previously (with respect to the new delete operation) issued operations have completed. But this element is not yet known. However, the numerical value of the entry to be brought at the root (and then drop down) is *not* important: any entry can be brought up, as long as entry swapings are made so as to end with a correct heap. For this we can use *'the higher element of the ones that are currently flowing towards the bottom of the tree and accompany an insert operation'*. This element is not necessarily the one that would become the rightmost leaf, as it will be subject to comparisons on its way to the bottom. However by stopping it from its way we know that we stop the creation of the rightmost leaf. We furthermore leave the tree's invariant condition intact, as the currently stored elements do not violate it. Only the flowing elements associated with insert's have not already obtained a correct position. The element just extracted will do for the new root, as from there it will again try to find its correct position into the tree. Taking the element that would indeed become the rightmost leaf would not make any worthwhile difference.

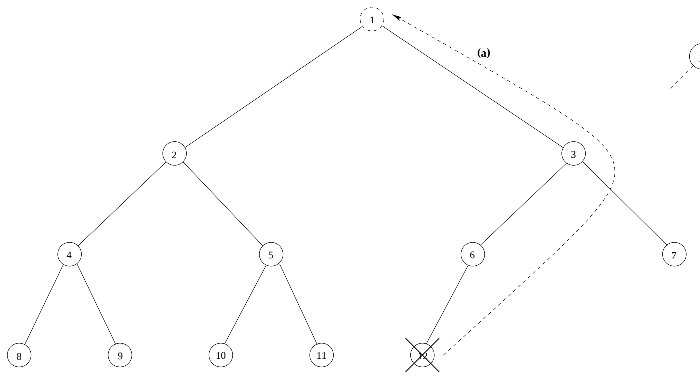

Figure 8: A delete operation when no insert's are pending

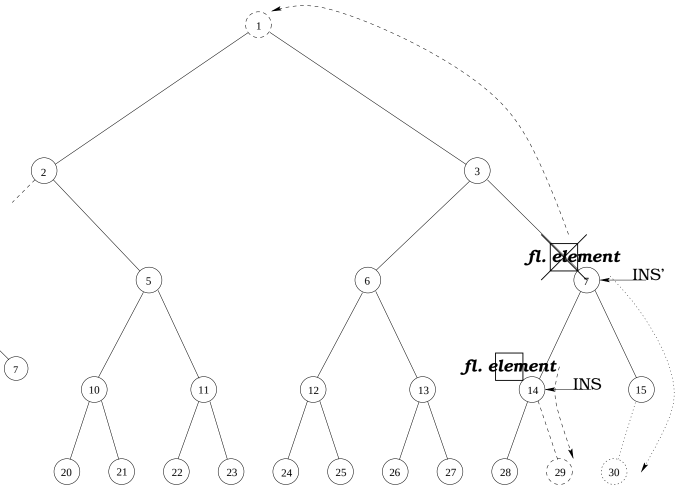

Figure 9: A delete operation interacting with the topmost insert

4.3.2 Overlapping the Operation of Successive Stages

Delete (or replace) operations on a node i, in each stage of figure 7 , 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.

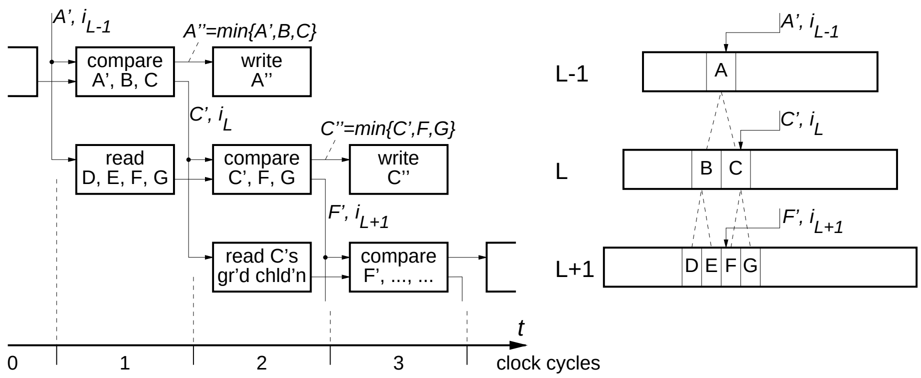

Figure 10: Overlapped stage operation for replace

We can improve on this rate by overlapping the operation of stages. figure 10 shows delete (or replace) –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 –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[6]; 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 will be further analyzed in section 6.1 . More details on how the overlapping of successive stages is performed, and how this is used to achieve a high operation throughput, are thoughroughly presented and analyzed in section 5 .

4.4 The Ability To Manage 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 7 , 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

[6] 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.

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.

5 Pipeline Architecture and Alternatives that were Rejected

Every operation performed on the tree and beginning from its root, "falls down" towards its leaves with the speed of one level after a number of clock cycles. Our concerns are (i) the speed of "falling down", and (ii) the rate at which operations can be issued, that is the time interval after which a new operation can follow the previously issued one. These rates will determine the operation throughput that can be maintained by our design. As the replace operation is very similar to DEL we will only deal with the DEL and INS instructions. Therefore, 4 possible pairs are to be examined: INS-INS, DEL-DEL, INS-DEL, DEL-INS.

5.1 CPI = 4 without Bypass Logic

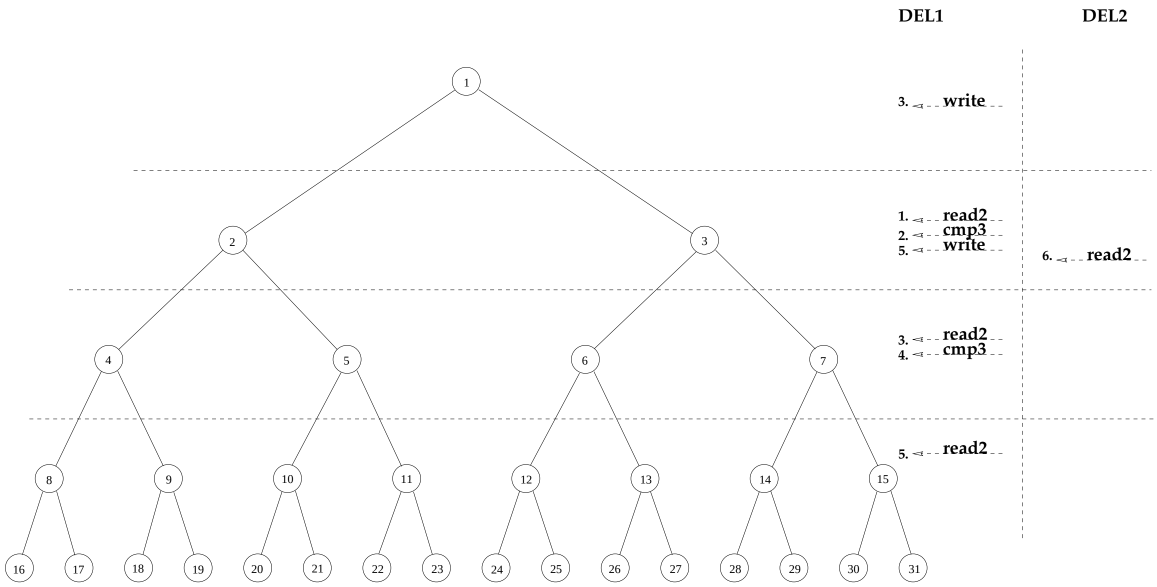

Figure 11: The simple approach scheme dealing with a DEL following a DEL

We will start with a simple thought that uses no bypasses. Starting with figure 11 , we can firstly see how a delete operation evolves. A read[7] operation for the children of the root is issued at clock cycle 1. Then these two along with the new root element

[7]the notation read2 means that 2 elements are read. In the following discussion we will also see a read4 notation, with analogous meaning

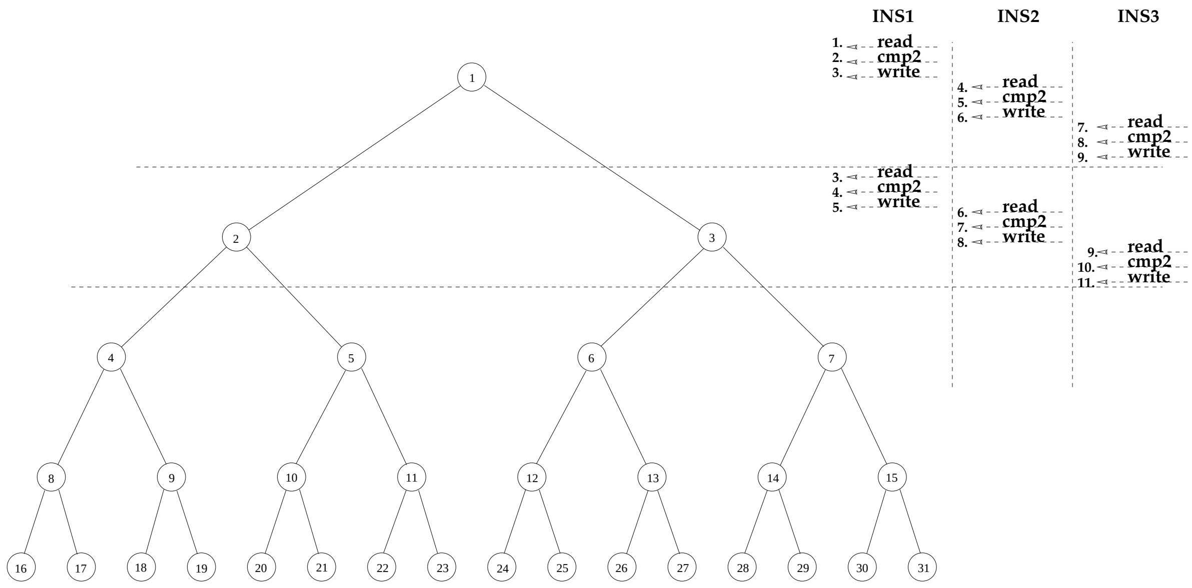

Figure 12: The simple approach scheme dealing with an INS following an INS

are compared in cc 2 and the smallest of the three is written as the new root in cc 3. If the root was already the smallest element, the process stops and the delete operation is complete. Otherwise the root is exchanged with the smallest child and the process continues for the node that now holds the initial root. Actually the root is not written at that node but it is rather "flowing" next to it as it will probably have to move again, making such a write useless. While the new (or old root) is written in cc 3, the children of the smallest node, already located through the comparison in cc 2, are being read as they will probably be needed for a forthcoming comparison (in the case that the delete operation is not completed yet). This comparison will take place in cc 4 and the write of the smallest element will take place in cc 5 (along with the next read). Finally in the same figure we see that the next delete operation can be issued in cc 6, after the children nodes it wants to read are already written.

In figure 12 we can see how the insert operation evolves. The newly inserted element can be though of as flowing next to the root just after the operation is issued. In cc 1 the root element is read, and in cc2 it is compared against the flowing element. In cc 3 the smallest element of the two is written as the new root, and the newly flowing element is now "placed" next to one of the children nodes, chosen in the manner described in section 4.2. While the new root is written, the chosen child node is read in parallel for the next comparison to take place in cc 4. The operation traverses the tree downwards, acting recursively, until the element that results after all the comparisons is placed at the leftmost empty leaf. As shown in the figure, a next insert operation can be issued in cc 4, right after the root it want to read is already written.

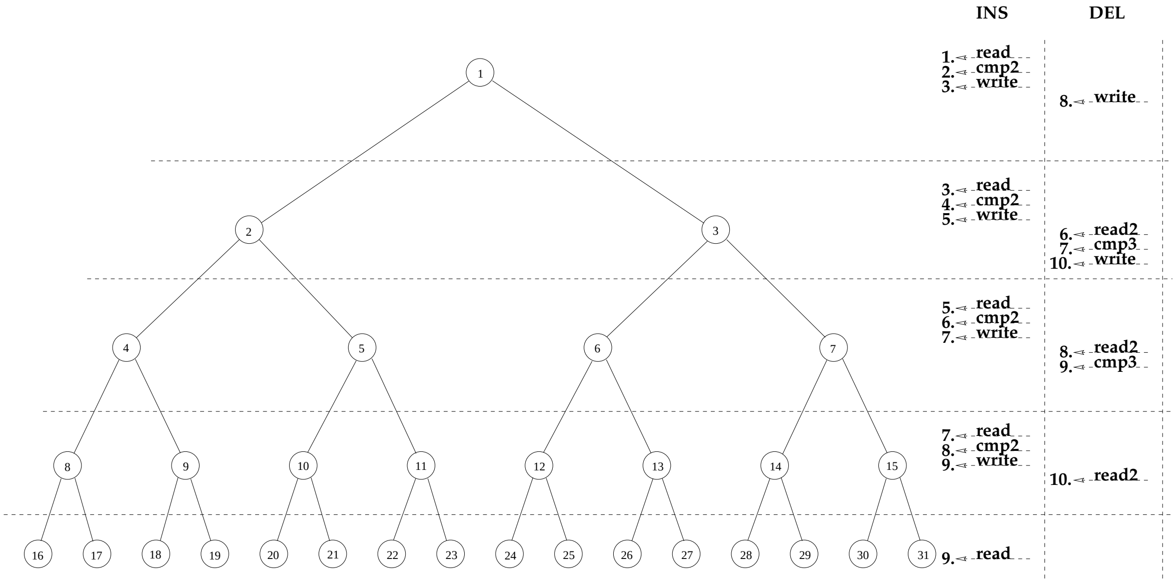

Figure 13: The simple approach scheme dealing with a DEL following an INS

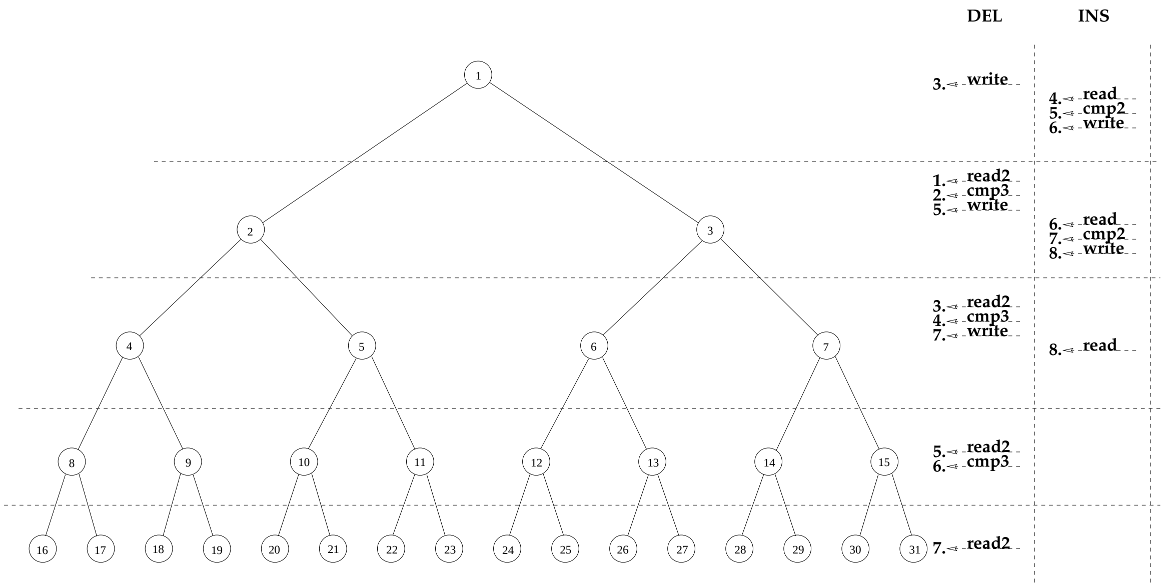

Figure 14: The simple approach scheme dealing with an INS following a DEL

Finally in figure 14 and figure 13 we can see how an insert or a delete operation can follow each other. In figure 14 we can see that an insert operation can follow a delete operation in cc 4 (leaving two cc's idle), as it has to wait for the new root to be written. In figure 13 we see that a delete operation can follow an insert operation in cc 6 (leaving 4 idle cc's), as it has to read the children of the root, written by insert in cc 5.

Although this approach is very simplifying, it gives us insight of how each operation evolves through the tree, and of the data dependencies that are encountered. Speaking of performance we see that each operation traverses the tree with the speed of one level per two clock cycles. With this speed, the best operation throughput we could achieve is one new operation every two clock cycles, if we want each operation manipulating distinct tree levels. However if we consider delete and insert operations each constituting the 50% of all the operations, then each new operation is issued every 4 cycles in average, thus leaving 3 cycles idle after each new issue.

5.2 CPI = 3 Using Simple Bypass Logic

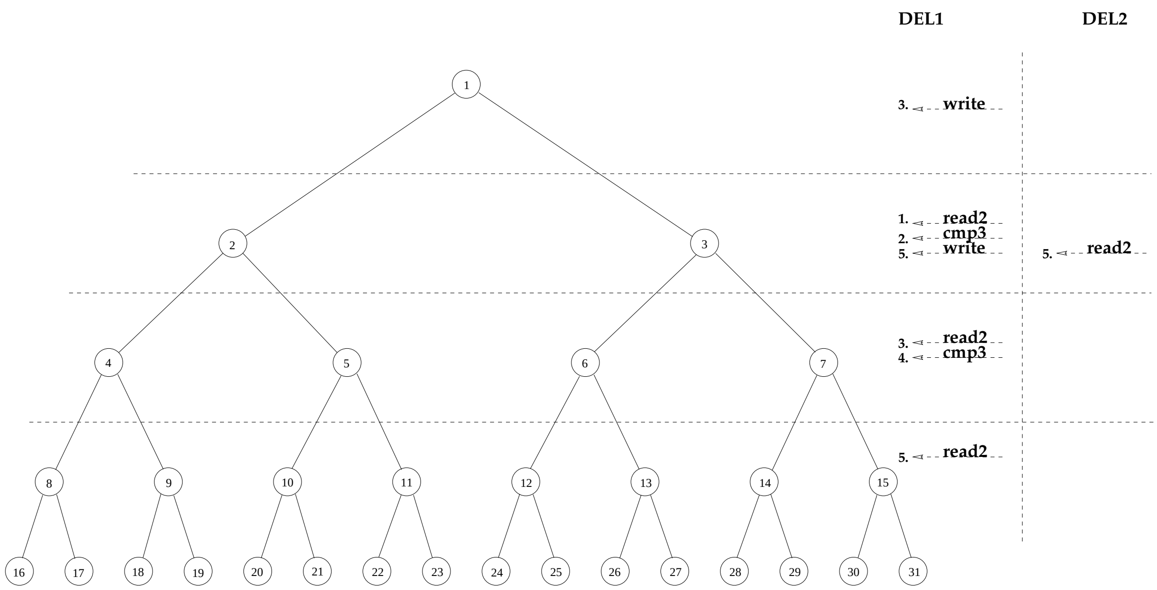

Figure 15: A DEL following a DEL with usage of bypass logic

After seeing the simplistic approach, some improvements are intuitive. In this section we see the improvements gained by the introduction of bypass logic. The new timings are depicted in figure 15 , figure 16 , figure 17 and figure 18 . The bypass introduced here concerns the issuing of new operations. Each time a new operation is issued it wants to read one or two elements that are written by the previously issued one. Instead of having to wait for the elements to be written, they can be forwarded by the corresponding

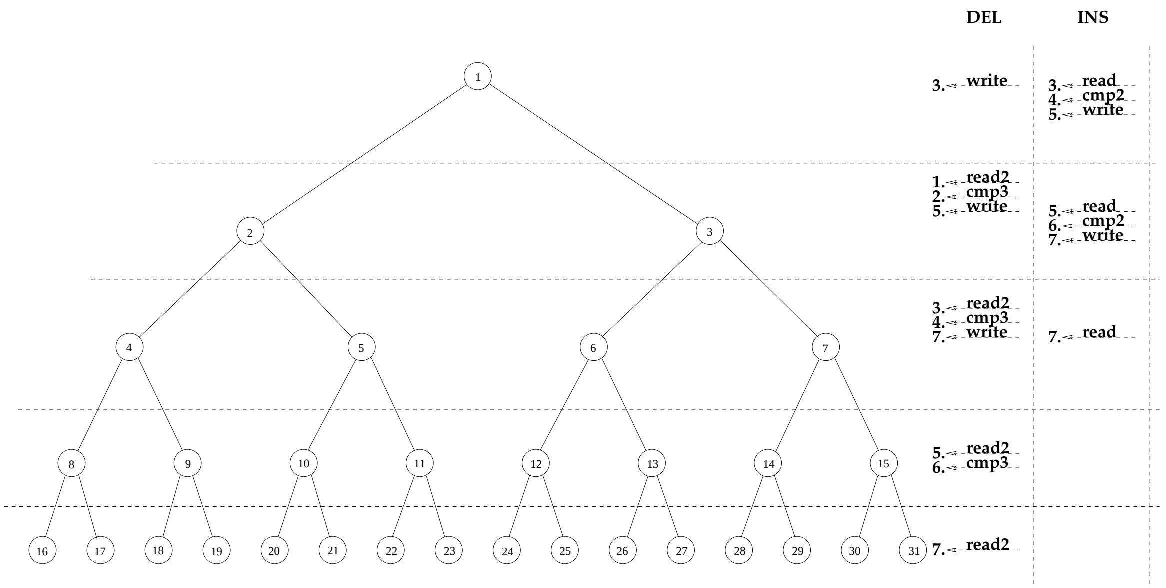

Figure 16: An INS following a DEL with usage of bypass logic

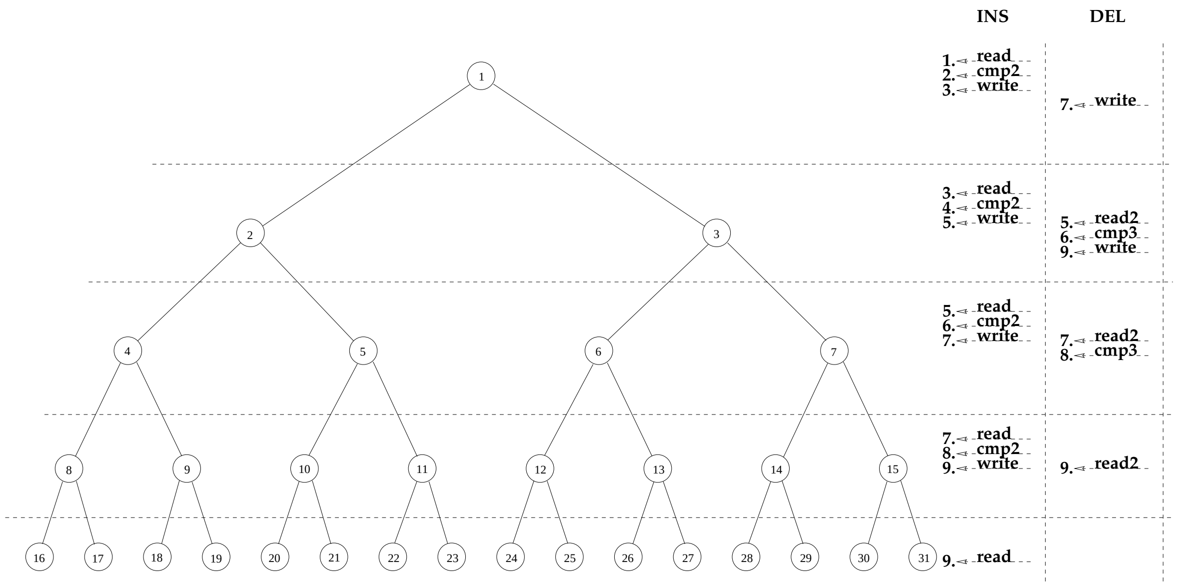

Figure 17: A DEL following an INS with usage of bypass logic

compare operation which decides the new elements, and the read operation can proceed in parallel with the write. In that way a new delete operation can be issued in cc 5, while a new insert operation can be issued in cc 3. This results in an aggregate throughput of a new operation every 3 cycles, considering the simplifications of operation percentages discussed earlier. Thus, now, 2 cycles are left idle after a new issue.

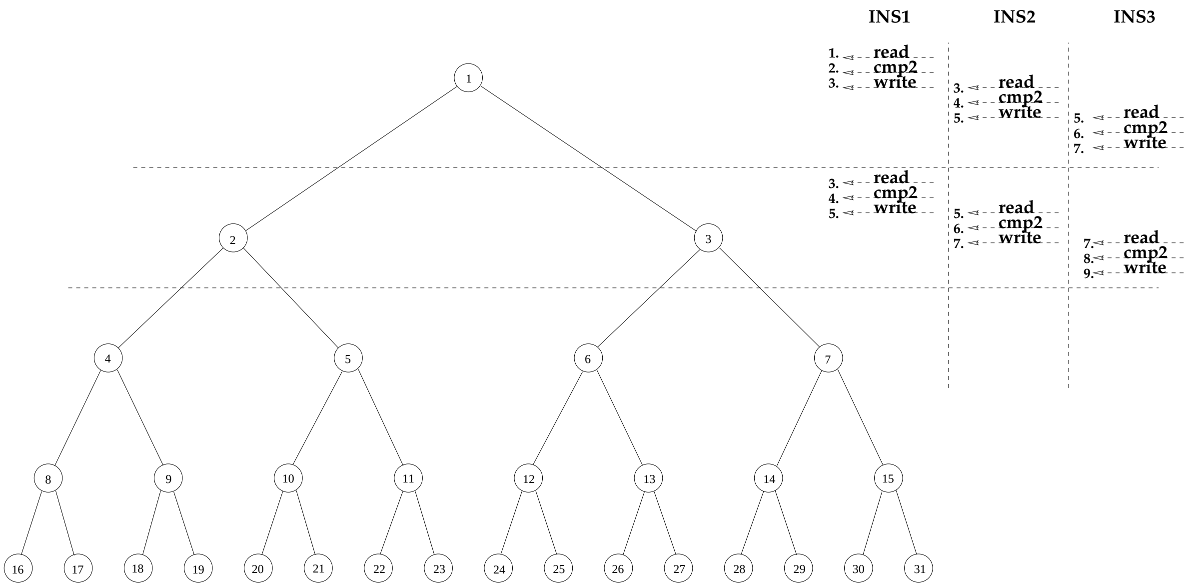

Figure 18: An INS following an INS with usage of bypass logic

5.3 Ripple-down Rate of 1cc/level, leading to CPI = 2.5

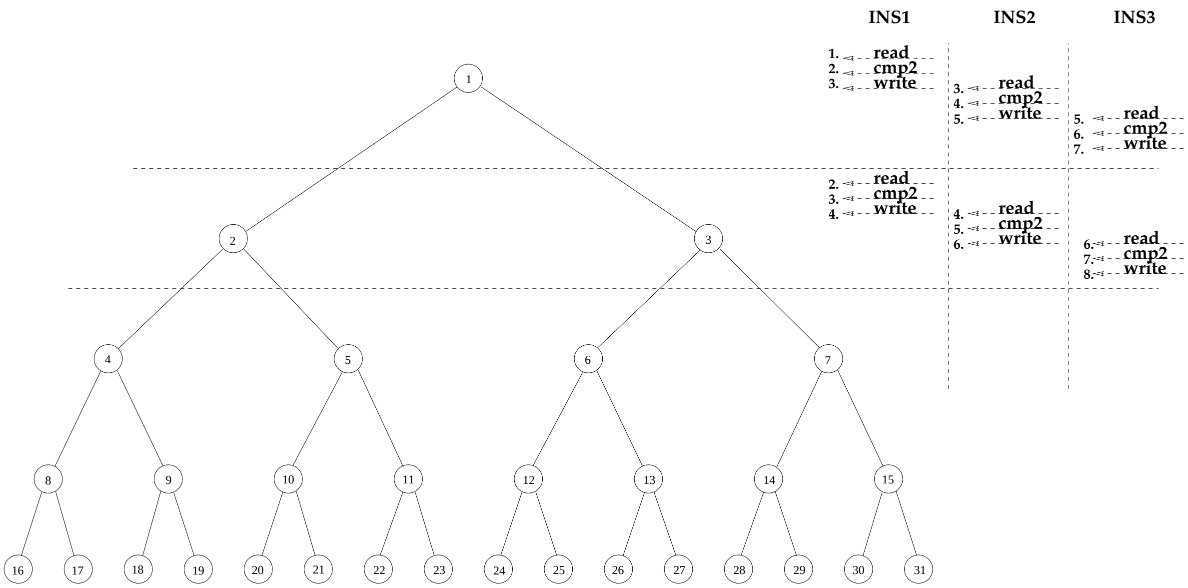

Figure 19: The aggressive approach scheme dealing with an INS following an INS

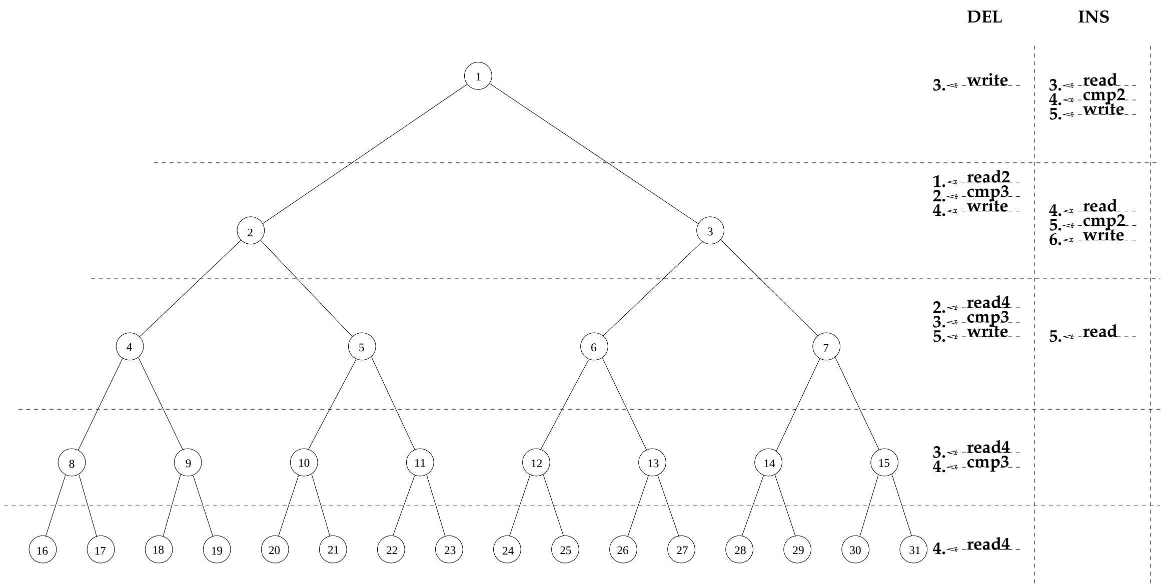

Figure 20: The aggressive approach scheme dealing with an INS following a DEL

Here we present some ways of making the design more efficient. We saw earlier that the operations traverse the tree at the speed of one level every two clock cycles. If we want performance to improve significantly in respect to the design using bypass logic (section 5.2) we should improve this speed. Furthermore both operations should be accelerated: making only one of them faster would not lead to any improvements – see section 4.3.2 – because the slower operation would stall a faster one behind it. The improved timings making both operations faster are depicted in figure 19 , figure 20 , figure 21 and figure 22 . We will start with figure 19 which shows the new timing for the insert operations as this change seems more straightforward. Here the difference is that although the first level of the tree is dealt with in the same cycles, the read operation for the next level is issued in cc 2. This is feasible because the entire route towards the bottom is known from the beginning for an insert operation, so the new node to be read is also known. The following comparison in cc 3 will have to take the newly read element and the greater element resulting from the comparison of cc 2. A next insert can be issued again in cc 3. However a next delete operation, as shown in figure 21 , can be issued one cycle earlier, in cc 4, because the write of the element for the insert at the child of the tree root is taking place one cc earlier, and the same goes for the first read of the delete operation that takes the latter by forwarding.

In figure 22 we can see the timing of the accelerated delete operation. Here to make the speed of traversing the tree equal to one level per cc, we should again perform a read on each level before the comparison at the previous level has completed. However this is a problem with delete, as the comparison not only chooses the element that will

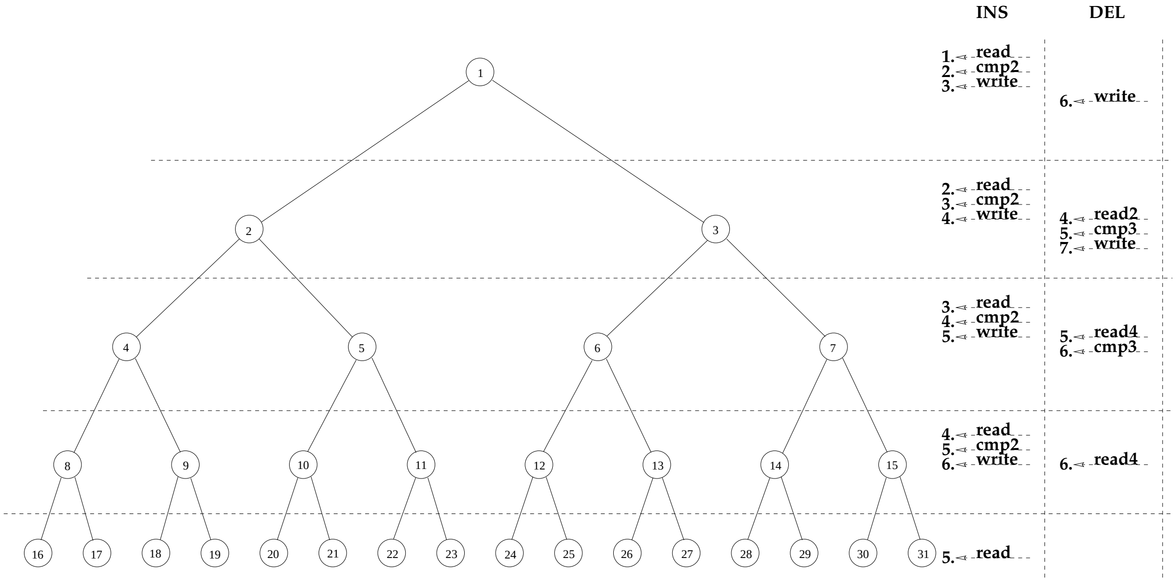

Figure 21: The aggressive approach scheme dealing with a DEL following an INS

continue flowing, but also the way to be followed towards the leaves. If we don't know the outcome of the comparison then we don't know which elements to read. The solution to this problem is reading all possible four elements, as mentioned in section 4.3.2 . Then, and after the outcome of the comparison is ready, we will know which of the four elements just read will be used for the next comparison. Since the previous comparison has been resolved at the time a new one starts, three elements will always have to be compared. A next delete operation can now be issued in cc 4, while a forthcoming insert operation, as shown in figure 20 , can be issued in cc 3.

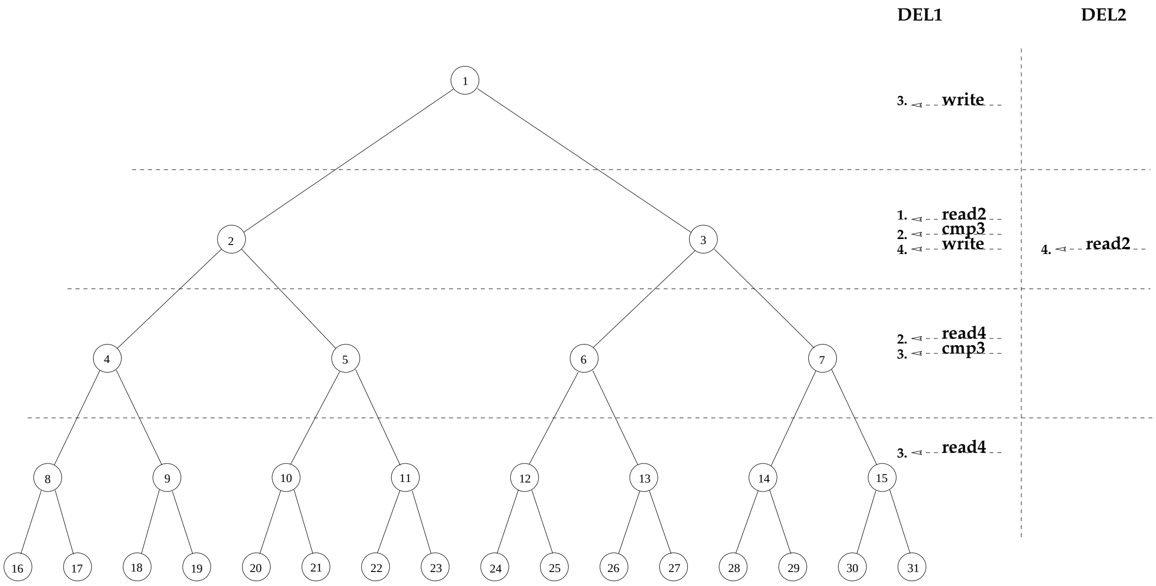

Figure 22: The aggressive approach scheme dealing with a DEL following a DEL

This aggressive approach results in both operations having the traversal speed of one level per cc and the average operation throughput maintainable is now one new operation every 2.5 cycles, with 1.5 cycles being left idle between operations, on the average

5.4 More Intensive Bypassing for CPI = 1.5

In this section we present the version that was subsequently synthesized in detail. It incorporates all previous mechanisms and furthermore adds more aggressive forwarding, ending up in a much more efficient design. The corresponding timings are depicted in figure 23 , figure 24 , figure 25 and figure 26 .

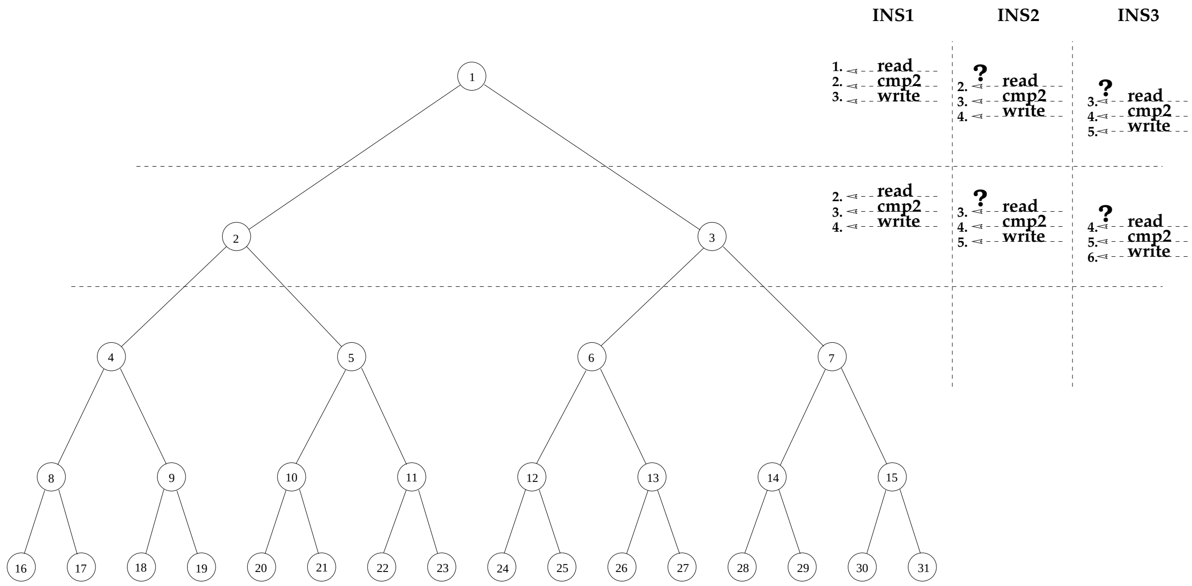

Figure 23: The timing for an INS-INS sequence with our more efficient and implemented technique

We will start with the insert operation depicted in figure 23 as it seems more straightforward. In this diagram we see that the second insert operation is issued right after the first one in cc 2. At that time, INS2 reads the value it needs before this value is even decided by the comparison at cc 2. However if we are careful this issuing can work correctly without any need for stalls. This is true as the INS2 actually needs the value read for the compare operation at cc 3. At that time the value is known as the comparison of INS1 is executed at cc 2 and its result can be forwarded. However the read operation is necessary, because, as we traverse the tree downwards, two successively issued insert operations can take a different turn. From there on the value to be written by the first one is independent of the second one, and thus the second insert operation can and must read the element it needs for itself. This uncertainty of whether the read operation is useful or not is indicated in the diagram by a question mark over the read arrow. A similar situation is depicted in figure 24 where an insert operation following a delete is again issued in cc 2 with the value for the compare at cc 3 forwarded from the cmp3 operation, taking place at cc 2.

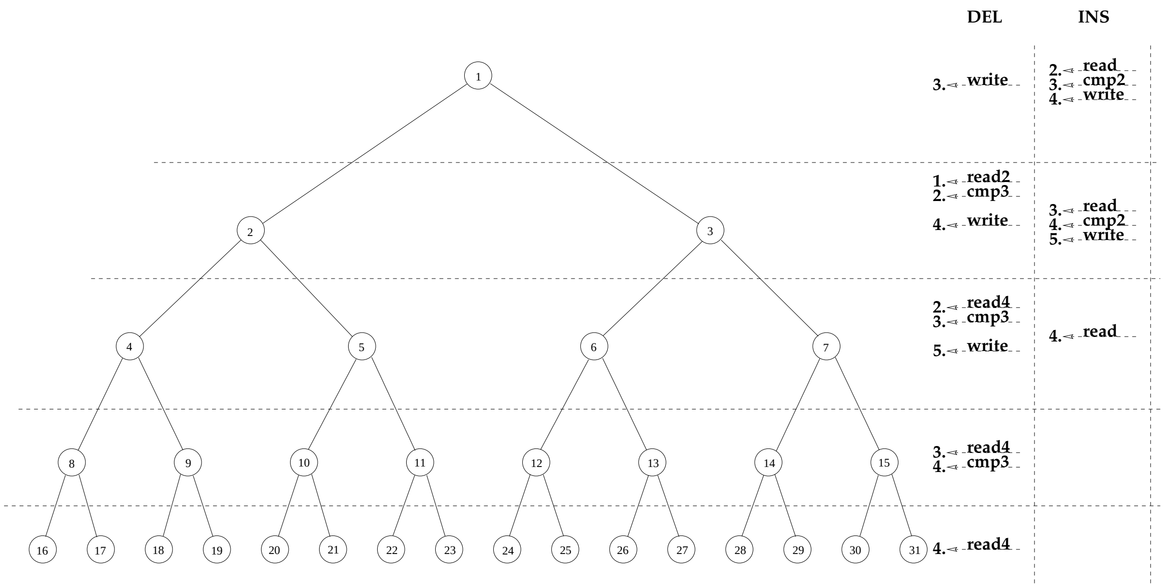

Figure 24: High performance issuing of a DEL-INS sequence

In figure 26 we can see the improved timing for the delete operation. The second delete is now issued in cc 3, while in the previous timing of figure 22 it was issued in cc 4. Thus the read operation is again executed early. However the cmp3 operation can still have its inputs valid at the beginning of cc 4. The root has already been written in cc 3. One of the children is probably being written in cc 4 (if it had to be exchanged with the root) but can be forwarded after the cmp3 operation at cc 3. The third element (the second child) was correctly read in cc 3 by the read2 operation. If after going down some levels, the two successive delete's have taken a different turn, then both of the children values will be taken from the corresponding read4 operation. However, for two successive delete's, the different turn does not make these two independent right away, as was the case for successive insert's. As each delete manipulates simultaneously two levels of the tree, one more level traversal (actually 1 cc) will be needed before the two operations are independent. In figure 25 we see the case for a delete following an insert. The delete is again issued in cc 3, and the correctness analysis is as before.

After these improvements, we have an insert operation following its previously issued operation in the next cycle, without any stall, while a delete operation is being issued with only one cycle left idle after either kind of operation. This gives us a (simplified) average operation throughput of one operation every 1.5 cycles with only 0.5 cycles left idle.

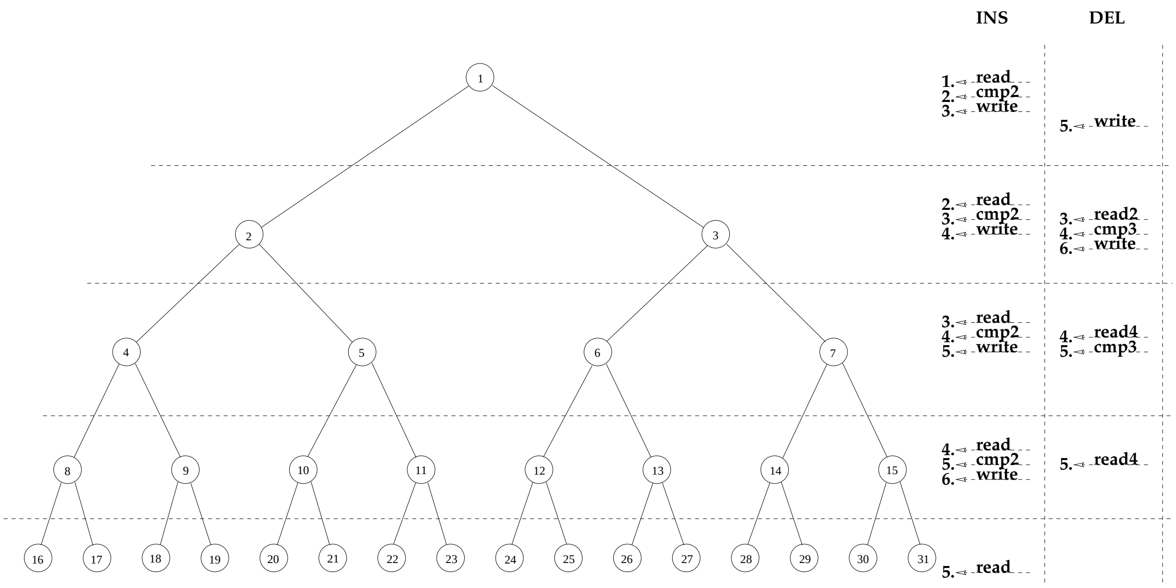

Figure 25: High performance issuing of an INS-DEL sequence

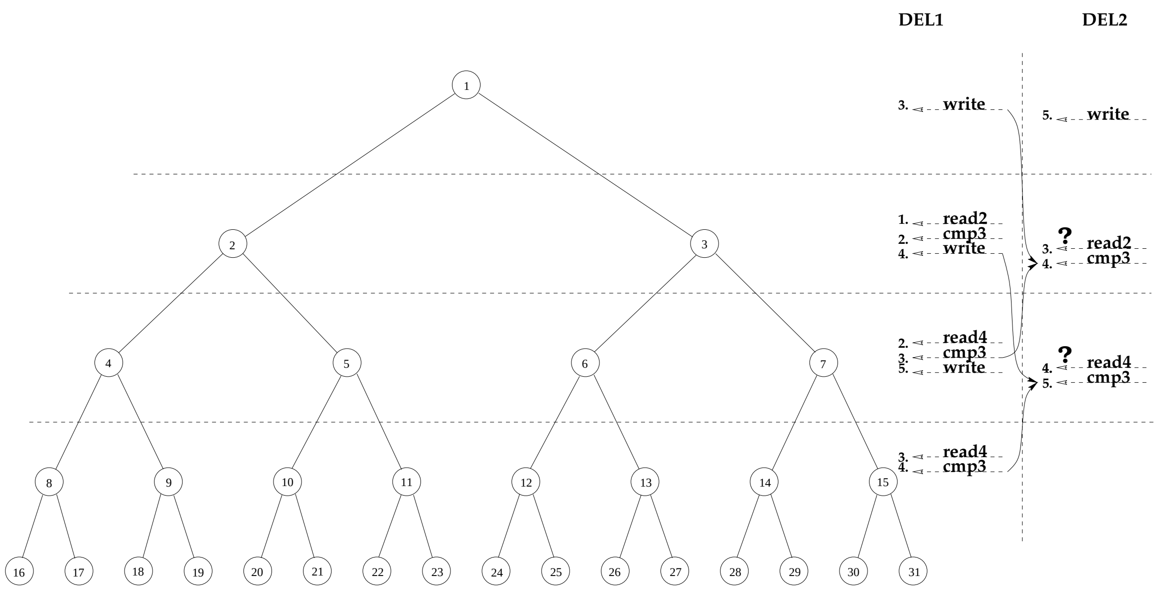

Figure 26: High performance issuing of a DEL-DEL sequence

5.5 Combining INS-DEL into idle-DEL for CPI = 1 to 1.25

The higher performanced we reached up to now, that of section 5.4 , is calculated with the thought that an idle cycle should always preceed a delete operation. Based on this thought, when we want to issue a command sequence like INS-INS-DEL, it should be issued as INS-INS-idle-DEL. However, for the reasons already explained in section 4.3.1 , the youngest INS will have to be canceled when the DEL is issued. After this, the command sequence INS-idle-idle-DEL will be left flowing through the pipeline. The two idle cycles seem too much now, as even a single intervening idle cycle would suffice, causing no problem to the pipeline. This gives us the idea to allow a DEL to immediately follow an INS (with an added idle cycle required only for a DEL-DEL sequence, transforming it to a DEL-idle-DEL), as we are certain that the preceding INS will be canceled, producing a bubble (idle cycle) flowing through the pipeline, ahead of the DEL. So, with this enhancement, we have the DEL requiring an added idle cycle only to separate it from an immediately preceding DEL. The calculation of the resulting average throughput now depends a lot on the context: a scheduler that has knowledge of this mechanism will probably always be able to schedule an intervening INS to separate DEL-DEL sequences, transforming them to DEL-INS-DEL sequences. These can be issued with no additional idle cycles, leading to the issue rate of 1 op / cc. On the other hand, if we assume an oblivious scheduler, and with the simplified operation percentages used before, we can assume that a DEL will be preceded (at full utilization operating conditions, i.e. always having a new operation to issue) either by a DEL or by an INS with equal probability. So the probability to have a DEL-DEL sequence can be given as $P(DEL \cap DEL) = P(DEL) \times P(DEL) = 0.5 \times 0.5 = 0.25$. Thus an idle cycle will be added every four operations (before every other DEL) in average, giving a throughput of 4 operations every 5 cycles, or CPI = 1.25.

Although the above looks simple, a problem must be identified and solved. When a DEL immediately follows an INS, we said that it will cancel the INS, leading to the creation of the needed bubble. However, as the serial issue of operations dictates, the DEL should have the same results as if the preceding INS has normally finished. This means that the pipeline structure used inside should not have side-effects to the outside. However, as we will see in section 7 , when this INS is canceled, it has not yet written the new root (if so was needed). In that case, DEL will output a (probably) stale root. The correct item will be outputed by the next DEL, leading to the interchange of two successive values of the correct output sequence. So the DEL should not directly output the root, but instead the smallest of the root and the canceled item. It will thus always correctly output the smallest of all items, leaving the unselected of the two for the root substitution (possibly being the old root, but this causes no added bothers, as it does not differ from the general case).

5.6 An Alternative Schematic Description of the Pipelined Execution

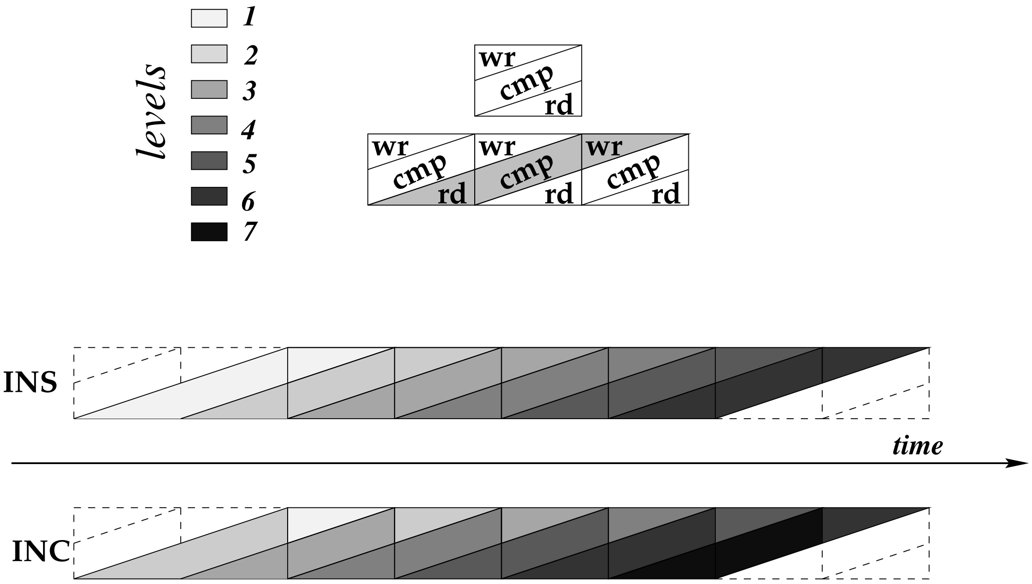

Figure 27: A Depiction of the Pipelined Execution of Operations

The way the pipelined execution of operation proceed can be nicely presented through figure 27 . In this figure each rectangle, as the one depicted at the top, is divided into three parts, each corresponding to one of the three possible actions; read, compare and write. Each such rectangle corresponds to a clock period. Bellow it, in the figure, we see three such rectangles placed beside each other representing three successive clock cycles. The highlighted stripe that runs from bottom-left to top-right represents the actions executed by an operation at a distinct tree level. So, as time advances from left to right we see that in the first cycle the operation performs a read, in the next cycle a comparison is performed at the same tree level and at the third cycle a write is performed at that level. So we move along to the next two patterns where we can follow whole execution of an insert and a delete as each goes down the levels of the tree. Each different shading corresponds to the actions on a different tree level. In any clock cycle, all three actions are executed in parallel to achieve high pipelined performance, however each at a different tree level. Notice that for delete the aforementioned stripes include a *rd* and a *cmp* at the same level, while the *wr* occupies the level above. This is so as delete reads the children at level i, compares them against their parent and writes the new parent at level i-1.

5.7 The Implications of Issuing Every New Operation Per Clock Cycle

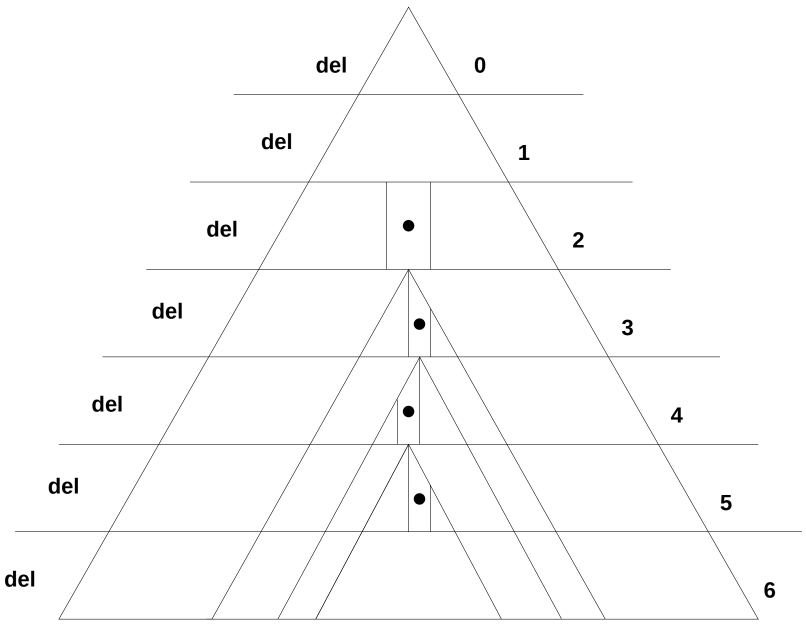

Figure 28: Issuing DELs without stalls leads to the need for added H/W complexity

Although up to now we mentioned that a delete operation cannot follow a previously issued delete without a stall, this is not the entire truth. As a matter of fact, added hardware resources can in effect solve this problem. The problem that actually arises with the continuous issuing of delete operations is illustrated in figure 28 . Let us think of a delete operation having to decide the new node at a specific level, say level 2, in a specific clock cycle. It has to compare the children of the node and the node itself to each other, and the minimum of the three is to be placed as the new node. However if this delete had been issued just after another delete, without an intervening idle cycle, the comparison could be very expensive. The previous delete, at the same time, is trying to determine a new node value for the next level of the tree, level 3 in the example. Furthermore, the new value of the node to be determined may be one of its children. If this is the case the delete at level 2 (referring to figure 28) should indeed not only compare the node and its children at level 3, but it should furthermore take under consideration (comparison that is) the level 4 children nodes of one of the level 3 children (the one to be decided by the level 3 DEL), as the parent at level 2 cannot be decided otherwise. The same is true all the way down to the leaves of the tree, assuming that successive delete operations have been issued. We finally end up with the need for very large comparators expanding across the whole tree and having a very high number of inputs (proportional to the number of levels). This added complexity is very high to pay in circuit area and design complexity.

What is more the benefit does not seem so worthy to justify such a design[8]. We therefore decided to exclude such comparators and demanding a stall cycle between two successive delete operations.

6 Cost-Performance Tradeoffs

In this section we discuss some alternatives concerning the memory width versus operation issue rate. We will present some memory parameters that may be varied in order to achieve a suitable overall design and we will indicate the corresponding cost/performance trade-offs. We start our discussion by examining varying memory widths for on-chip memories, and then we move along to on-chip vs off-chip memory.

6.1 Memory Width Alternatives

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 section 5.7 , 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* [9]. To have a concrete notion of memory width in mind, in our example implementation (section 7) each heap entry is 32 bits –an 18-bit priority value and a 14-bit flow ID; thus, dual-ported, 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 sections 5.4 and 5.5 and noted in line *(ii)*. In many cases, this will be an insignificant performance penalty, because we 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 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

[8]not to mention that clock speed would probably drop, leading to uncertain performance effects.

[9]More details on how these results were figured out and can be achieved by a heap manager can be found in appendix A

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.

local bypasses suffice.

The next step towards more narrow memories is to have $2 * 32 = 64$ bit 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. By reducing memory width to 2 entries, only 2 elements can be read in parallel. Insert operation remains unaffected as it did not need to read more than one element in a single clock cycle anyway. However, when we discussed how to accelerate delete's in order to traverse the tree at the speed of 1 level/cc (section 5.3) we saw that 4 elements should be read in parallel. Therefore this method can not be deployed and delete's will traverse the tree at the speed of 1 level every 2 cc. We are also not able to replace a $Rd4_{(i)}$ (read 4 elements command at cc i) with the sequence $Rd2_{(i-1)} - Rd2_{(i)}$ as at time i-1 we do not even know which four elements could be the candidates for the two children to be used. This happens as the compare_3 operation executed at time i-1 has not yet decided the grandparent of the elements at the level the reads are to be issued. Thus at time i-1 we could only have the information for a Rd8 command. Insert's will also need to be decelerated down to this speed in order to retain the same ordering of operations as they go downwards the tree. However, for both operations, we can use the technique of section 5.4 concerning earlier issued reads (issued before writes that should rightfully be issued previously) in order for new operations to be issued one cc earlier. 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.

When lower performance suffices, single-entry-wide memories can be used, reducing the ripple-down rate to 1 level every 3 cycles. The analysis follows the same fashion with the previous discussion with insertions being able to remain unaffected by the change, but with delete's now being able to traverse the tree at the rate of only 1 level per 3 cc, thus also hindering insert's. The $Rd2_{(i)}$ used with double width memories should now be replaced with a $Rd1_{(i)} - Rd1_{(i+1)}$ sequence. So, with local-only bypasses, deletions cost 4 cycles (line *(vi)*), while 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. Results for even narrower memories can be easily calculated, but are not presented, because performance degrades too much for them to be significant.

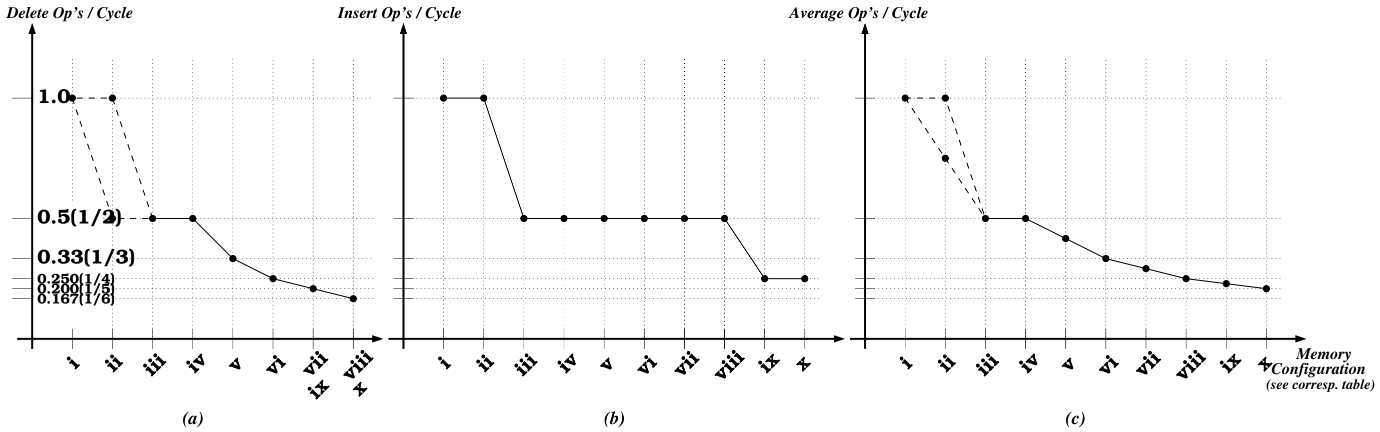

Figure 29: How Performance Degrades With the Use of Less Costly Configurations

6.2 Deploying Off-chip Memory

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[10], 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

[10]single-ported, of course

(ZBT)[11] 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.

figure 29 shows how the performance degrades as one goes through the alternatives of *Table 1* . In figure 29 (a) the performance of delete operations is shown, figure 29 (b) shows the performance of insert operations, while figure 29 (c) shows the average performance, assuming that the same number of insert and delete operations is issued (i.e. 50% DEL & 50% INS). When dashed lines are used, they represent the uncertainty due to the varying issue time of a delete in the case of line *(ii)*. In that case the performance can be considered to be inside the borders of the polygon formed by the dashed lines.

7 Gate-Level Synthesis of the Heap Manager Prototype

We will describe the design in detail, starting with a general external description at the interface level. Then we will move on by presenting the datapaths and how some specific blocks function.

7.1 Interface of the Heap Manager

The external interface of the heap manager is depicted in figure 30 . The signal *cmd* is the command given to the module; insert and delete are the two possible commands, however the second bit is RFU. *Valid* input signals the validity of the command given at each clock cycle. The whole interface is synchronous, synchronized to the *clk_in* clock input and thus all signals are sampled at every positive edge of the clock. *Data_in* is the data given as input, if any is needed. This is driven at the same cc the corresponding command is issued. An id (e.g. VC) and a corresponding numeric value (e.g. priority) needs to be given for an insert operation. Delete, as expected, requires no input parameters. In our implementation the id is 14 bits wide and passed on the most significant bits of *data_in* while the corresponding numeric value (or just number for simplicity) is 18 bits wide

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

Figure 30: The Heap Manager outline

and passed on the least significant bits of *data_in*. *Reset* is needed for some necessary initializations at operation start-up. As far as the outputs are concerned, *reset_ack* signals when resetting is completed, i.e. all the needed initialization has taken place, and normal operation can proceed. This will take just 3-4 clock cyles. *Clk_out* is the clock signal synchronizing the output signals (which is synchronized to the *clk_in* clock), which are sampled at every positive edge of this clock. Finally *data_out* is the data returned by the manager. Insert expects no data as an outcome, while delete receives the root element of the heap through *data_out*, at the next cycle of the command issuing. An insert operation to a full heap and a delete operation to an empty heap will be ignored. Id and number are included in the 32 bits of *data_out* in the same way as described for *data_in*. One last thing to mention is than an insert operation can be issued at any clock cycle, while for a delete we need to have an intervening idle cycle if it immediately folows another delete command. This is the outcome of what has already been explained in the section presenting the pipelined execution of operations and the relative methods to accomplish high operation throughput.

7.2 The Datapath

The datapath of our design can be divided in two major parts. The first includes all the needed functionality for the insert operations while the second handles delete operations. As the two operations share almost no functional blocks, the datapath partitioning suits gently and is also helpful for both the understanding and furthermore the implementation of the design. The memory is shown in both datapaths, however there exist only one copy of it, used by both (by multiplexing the inputs and outputs). The two otherwise distinct datapaths communicate with each other through bypass signals and respective data, needed to accommodate the fairly intensive pipelining. These are identified by the names D2, D3, D4, etc. and I2, I3, I4, etc. in figures 31 and 32.

7.2.1 Datapath Portion for Insert

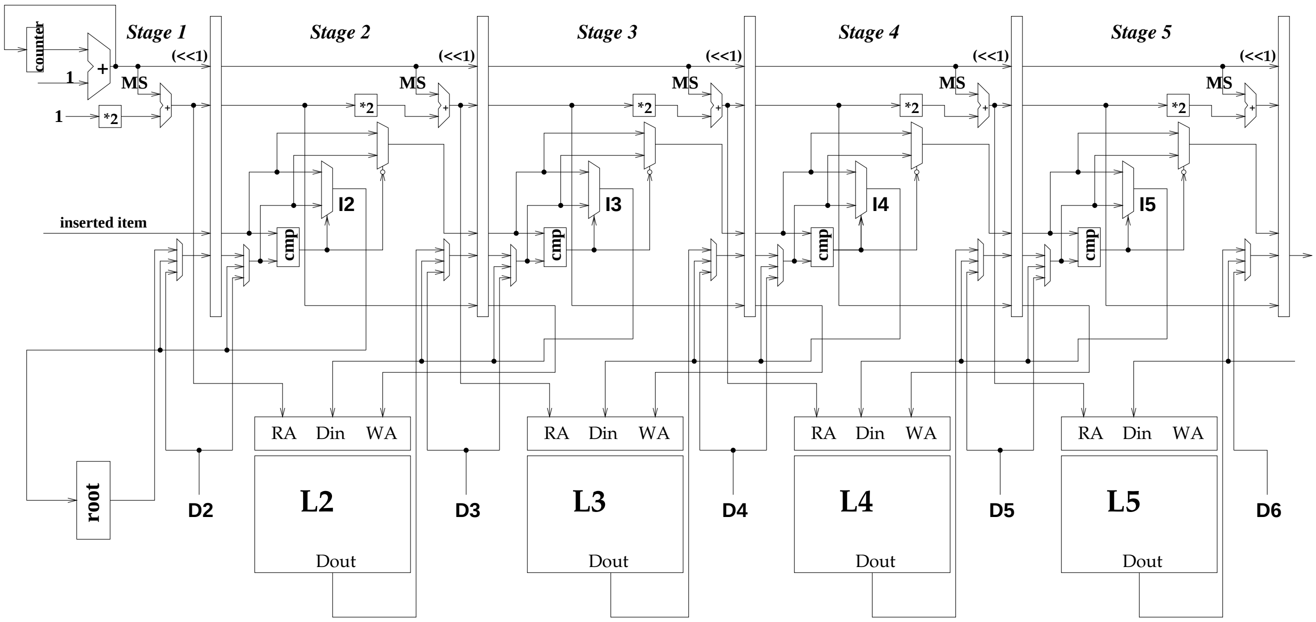

Figure 31: The datapath portion for the insert operation

The datapath for the insert operation is depicted in figure 31 . At first glance we notice that it is highly regular. One pipeline stage constitutes its primary part, being repeated uniformly. In that way the heap manager becomes easily size-configurable, with the appropriate number of stages repeated to form a manager of the desired size. Only the first two stages and the last one differ from the primary. The second and last stage are simpler variations of the primary stage. The first stage includes some extra functionality regarding the interface with the outside world, start up calculations and some useful data update and storage. Starting our description from this stage, we see a counter holding the number of elements currently stored in the heap. Each time an insert (or a delete) is issued, the counter is incremented (or decremented). A description of greater depth of the counter and its update mechanism is given later on section 7.4 as some extra techniques needing more analysis are deployed at this part. At stage 1 the root value is also passed on to the next stage. The root is kept into a register that corresponds to the single entry memory of stage 1. The item to be inserted is also passed on to stage 2 after being received through the interface. A last thing done at that stage is the calculation of the read address for the next stage. This address is calculated based on the value of the counter and the current read address (i.e. the address used to read from the memory corresponding to the current stage). The latter is 1, as this address corresponds to the root. As for control signals not shown, we ought to mention that a valid bit must be formed to signal whether an insert operation has been issued. This will be propagated on to the next stages and will contribute to the evaluation of oe, we and cs signals.

Next we describe the primary stage, which, being used as many times as needed, builds the heap manager of the desired capacity. We will not explain separately the 2^{nd} and last stages, which we reported as differing from the primary one, as they constitute simpler variations of the latter, intuitively comprehended. As we give the description the reader can refer to stage 3 of figure 31 . First of all we see at the left a comparator having two inputs. The first one (top) is the element flowing towards the bottom of the tree, while the second is the element currently stored at the previous tree level. The latter's path is determined by the path we have to follow to reach the empty tree position to hold the new element. However as other operations may be traversing the tree, this element, conceptually occupying the tree position, may not be able to be provided by the memory of stage 2, as this may contain stale data. A bypass path should then be activated to provide the data that are currently trying to be written into memory (bypasses are analyzed at section 7.7). The two muxes, one at the left of stage 3 and another at the right of stage 2, choose the appropriate input. Considering the inputs from the first (stage 2) mux to the second and taking each mux's inputs top to bottom, we have the following cases: (i) Take the data directly from memory. This happens when no previously issued operation, that involves a write to L2, is currently executing. (ii) An insert was issued previously[12] with an intervening idle cycle. (iii) A delete was issued previously with an intervening idle cycle. (iv) This is the input taken from the previous mux. (v) An insert was issued on the previous cycle. (vi) a delete was issued on the previous cycle. After having decided both elements, the comparator finds the one with the lower priority value. This is chosen by the first mux to be written to L2 memory at the address we previously tried to read from (as discussed above). The other element is given through the other mux on to the next stage, and constitutes the as called 'flowing element'. The new address to read from is also calculated. What is shown as a multiplier (*2) and an adder is actually implemented through simple bit manipulation.

7.2.2 Datapath Portion for Delete

The datapath for the delete operation is depicted in figure 32 . This datapath is also highly regular, however this can not be realized fully by the figure. This is so as the primary stage for delete is deployed from stage 4 and on. Thus we have the first three

[12] we refer to the time of issuing relatively to the time the operation currently under consideration was itself issued.

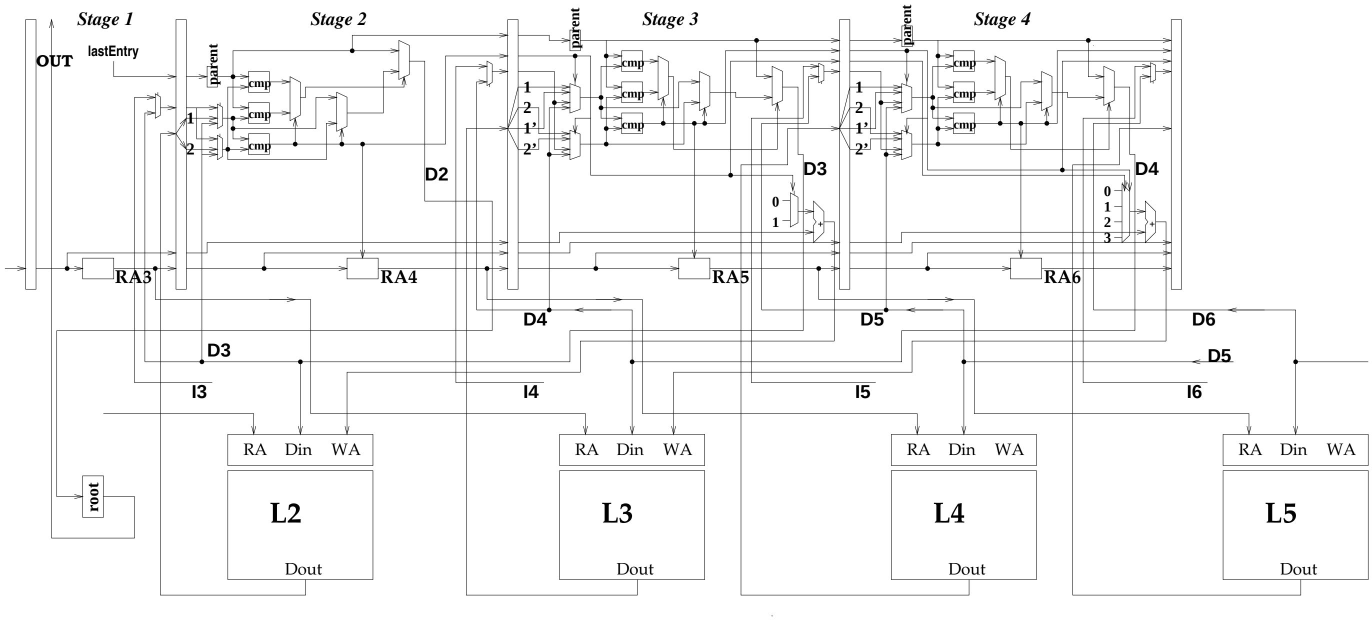

Figure 32: The datapath portion for the delete operation

stages constituting simpler variations of the primary stage. By repeating the latter as many times as needed, we end up with a heap manager of the desired size. The last stage is again a simpler variation. The first stage is the simpler of all variations, however it includes some extra functionality for the same reasons discussed for insert's stage 1. It reads and outputs the root of the tree, which will then be replaced. The new element to be (conceptually) placed temporarily at the root, as discussed in section 4.2 describing how operations work, will be given either by a 'canceled' insert or directly from the memory position it resides. This is simply shown through an arrow at the figure. The read address for L3 memory is also calculated.

Let us now describe the logic for stage 4 of figure 32 . This is a copy of the primary stage for delete, and through this we can also understand how the stages mentioned as its variations function. Four elements are passed from the previous stage, indicated as 1, 2, 1' and 2'. These are the four children needed to be read from the previous stage for the efficiency purposes mentioned when we discussed how delete executes (section 5.3). However by the time we reach stage 4, we know which two children are the needed ones. Nevertheless, bypass paths are candidates to give the actual values for the children, as the memory may have provided stale data. We have three cases needing a bypass path to be activated, corresponding to specific operation sequences. (i) The DEL was issued through the sequence INS-INS-idle-DEL. The second INS will have been canceled for the purposes of the DEL, however the first one is still active and tries to write the memory level we read from. (ii) A DEL follows this DEL with two intervening idle cycles. (iii) A DEL follows this DEL with a single intervening idle cycle. The corresponding multiplexors deciding which path should be activated are the 2-to-1 mux at the right top of stage 3 and the 4-to-1 mux at the left of stage 4. The bypass cases mentioned above follow the order of the mux inputs from left (stage 3) mux to right and from top to bottom input. After having decided the two children to be used, they are compared against each other and against their parent in order to find the smaller of all three. The first mux – after the comparators – chooses the result of the comparison between the parent and the smaller child. The smaller child is propagated through the next mux and the third mux chooses the smaller between the parent and the smaller child. The element chosen will be written at the appropriate memory address. We can calculate this address as we know the address used to read the four children and we furthermore know the results of the previous stages' comparisons. The mux in the figure selecting 0, 1, 2 or 3 and the adder are actually implemented through simple bit manipulation. Finally at the bottom we see the calculation of the new read address. This also needs just bit manipulation.

7.3 Dealing With Infinitely Increasing Numeric Values and Their Comparison

There exists a problem emerging which is associated with the numeric values giving order to the elements residing in the heap. As we have already stated, a scheduler may need to increase the number corresponding to an id so as to update its position in the current ordering. What is more this may be repeated an arbitrarily large number of times before the corresponding element is requested to leave the heap (through a delete operation). As this stands for all elements, we have monotonically increasing values, eventually leading to saturation of the number field of the element, no matter how many bits wide it is. A solution should be deployed to deal with infinitely increasing values and furthermore retaining the ability to correctly make comparisons among them.

Value wrap-around solves this problem. This technique works as follows. figure 33 helps our discussion. If we have an unsigned number of n bits then as the number increases continuously it will eventually reach its maximum value, $2^n - 1$. A consecutive increment will bring it back to the beginning starting from 0 and on again, or according to figure 33 fall down to the lowest level and start ascending again. However a number that just fell to to the lowest level should continue to be considered greater than a number residing to the top level (and has not been reset as many times as the former one). A normal comparison however cannot distinct between the case a number of the lowest level has just been reset and should be considered greater and the case the number resides there from start-up and is thus lesser. What we do to achieve a successful comparison is to allow the actual size of

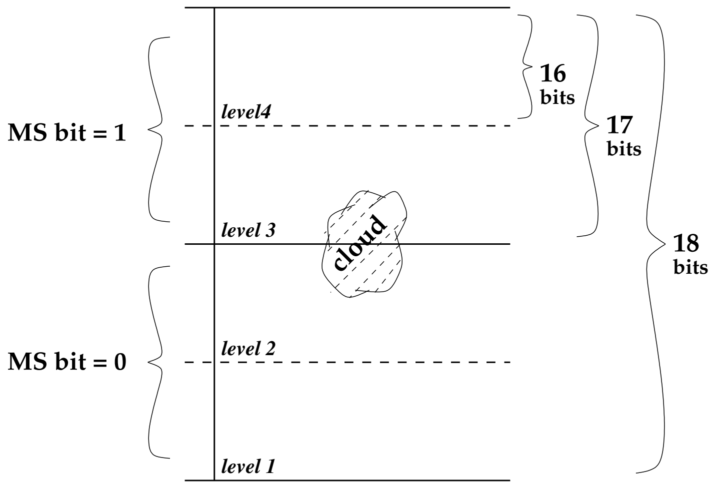

Figure 33: How continuously ascending values eventually lead to value wrap around. The value grouping resulting from our technique can be observed

the numbers to be only n-1 bits wide. In this way we initially have all the values residing at the two lower levels of figure 33 . If we furthermore allow the values to be incremented only by a n-2 bit value then our values will be allowed to spread only across two levels at most. As values increase the 'cloud' is continuously squeezed from two to just one level range and then this cloud moves round (bottom-to-top-to-bottom etc.) non-stop. At any time it spreads across two consecutive levels[13] at most. This is so as the increment value being only n-2 bits wide (and thus has a one level range) cannot move a number from level i to level i+2. Bare in mind that the number incremented can only be the smaller one, as for the heap we only want to increment the root element which contains the smaller numeric value. The comparison can now be executed meaningfully by considering a value residing at the top level being less than one residing at the bottom level.

A comparator's truth table can thus be formed easily and for a numeric value of 18 bit total width is presented in figure 34 . As we see the result can be given either by comparing the n-1 least significant bits of the number[14] (this happens when both numbers reside either at the two lower or the two higher levels of figure 33) or by checking only the two most significant bits (when these constitute one of the rest cases for the 'cloud').

7.4 Finding the Path for an Insert Operation

Insert has to place a new element adding a leaf at the leftmost empty position of the last not fully inhibited level of the tree. We can find the tree traversal from the root down to this position if we know the number of elements residing in the tree (including the one to

[13] the top level is adjacent to the bottom level

[14] which as stated above is the actual maximum allowable size of a number

Priority Comparator Truth Table

MS(A)	MS(B)	result
0X	0X	cmp 17 LS
1X	1X	cmp 17 LS
00	11	A>B
01	10	A<B
10	01	A>B
11	00	A<B

Figure 34: The comparator truth table for 18-bit total width numbers

be inserted), that is the updated counter. The algorithm goes as follows. First of all we need to have the counter of the heap in binary format. We take one by one its bits from the 2^{nd} most significant to the LS. Each time we find an 1 we go down a level turning right and move on to the next bit. If we find a 0 then we do the same but taking a left turn. We repeat this process until all the bits of the counter have been consumed. We will at that time have reached the wanted position. Although this algorithm is theoretically correct, in practice we come across some obstacles. The counter implemented by a register has a fixed number of bits, so finding the most significant bit quickly is not easy. A solution deployed to circumvent this difficulty is to have the counter actually placed at the most significant bits of the register. In this way shifting it left one bit at a time would provide us with the needed functionality. However we need some added information to point the number of the valid bits in the counter, in order to find the (now varying) position of the LS bits. Some zeroes coming after (right of) the rightmost ace could need to be included to the counter value, up to a point, while others from this point and on to the right should not.

A mask register can be added to offer this functionality. figure 35 depicts an example. Mask has leading zeroes up to a point, indicating the point where the last valid bit of the counter should be considered. Each time the counter expands its width (e.g. expanding from 111 to 1000) the mask is shifted right, thus moving the hypothetical validity point indicator one place to the right. In the opposite case that the counter is shrunk (e.g. updated from 100 to 11) the mask is shifted left with an ace entering its rightmost bit. To perform an addition to the counter we use all its "invalid" bits as aces by OR-ing it to mask, then we increment it by one and shift right both mask and counter if we have carry-out, thus satisfying the need for expansion. To revert the trailing invalid bits of

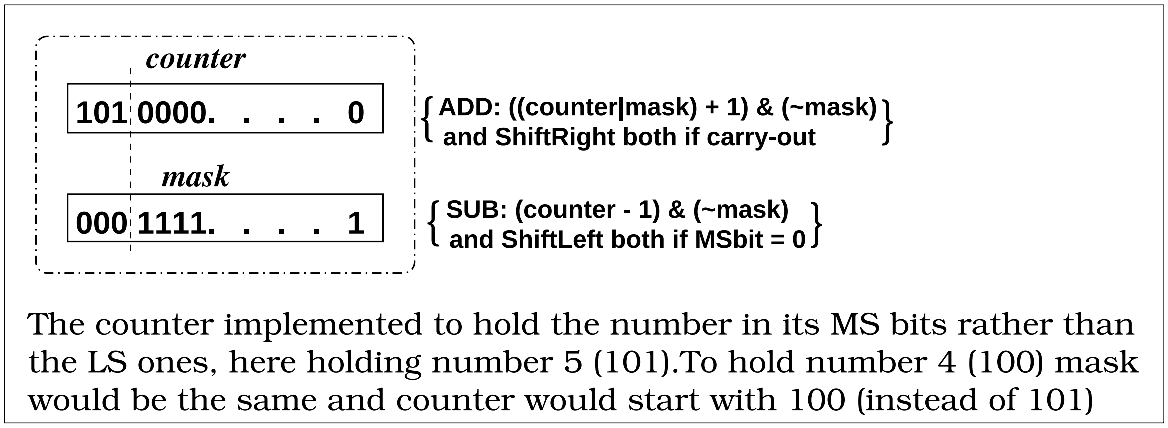

Figure 35: The counter and mask registers aiding to determine an insert's path

counter back to 0 we AND it with the bitwise-negated mask. The subtraction is executed somewhat more easily as seen in figure 35 , needing a left shift of counter and mask if it results to a leading zero at the counter. With the beforementioned way and the actions implementing counter increment and decrement we have the counter ready to efficiently supply the information needed to derive an Insert's tree traversal.

7.5 Managing Operation Completion

One issue emerging for the heap management is how operations reach completion, that is, how an operation understands that it has reached to its end and no other element rearrangements should take place. Let us start with insert. We already know how insert chooses its way downwards to the leaves of the tree until a new leaf is added. However when the lowest level is read, the element to be inserted is compared against the element currently residing at that leaf, which of course is 'null' and contains garbage. One possible theoretical way to deal with these comparisons (the last ones that is), for insert's as well as for delete's could be to initialize the tree with the maximum positive value for the number fields of all its elements. This would lead the comparison to choosing the flowing element, just as we wish. However our discussion on infinitely increasing values and the value wrap-around solution shows that no maximum number exists. So we must work around another solution for insert, which goes as follows. When we reach the last inhibited level, an information given by the up-to-then shifted mask, we enforce the corresponding multiplexor to choose the flowing element to be written to memory and the memory write enable signal to become valid. The operation is set to invalid thereafter.

For delete operation the situation is somewhat more complicated. The last comparison of a parent element against its two children needs to know how many of the children are indeed valid. This can be accomplished by computing level by level the (ascending) number of the children currently under consideration and comparing this number against the value the element counter had when the operation was issued. It should thus flow along the pipeline levels. If the number corresponding to a child is less than or equal to the element counter then it is valid, otherwise it is 'null' and should be considered to contain garbage. Based on the validity of the children, the comparison among the parent and its children is considered accordingly, giving no account to invalid children. Finally at the last inhibited level the parent, currently flowing, will be written and the operation will be set to invalid, thus being completed. Of course if the parent is found to be less than both its children at an intermediate level then it is written to memory at that time, the operation is set to invalid and the lowest inhibited level of the tree will not be reached at all.

7.6 Providing the Root Substitution Item for the Delete Operation

When a DEL is issued, then, as discussed earlier, the last item should be taken from the memory position it resides, replace the root and some repositionings will follow to keep the heap property intact. However if INSs are currently active in the pipeline, the last item cannot be located in a static memory position but is rather flowing dynamically into the pipeline. So we should take into consideration two different cases for each possible situation the pipeline could be at the time a DEL operation is issued; The case INS(s) are active in the pipeline, and the case that no INS (i.e. only DEL or "NOOPs") are currently active.

7.6.1 Insert Canceling

If INSs are currently active in the pipeline, the youngest one should be canceled, as this INS would be the one to eventually place the last item[15] we need. So a mechanism should be deployed to handle this canceling. The idea is as follows. Each time a DEL is issued, then if INSs are currently active in the pipeline, the youngest one is notified that it gets canceled. This INS selection is given by a priority enforcer with all the valid signals of the insert's datapath portion. The canceled INS will thereon be invalid and so will not continue to perform any actions in the next clock cycles. What is more, any writing to be executed from the specific stage of the pipeline (the stage the up-to-now youngest INS functions) will be halted. In this way the INS will not make any changes to the heap state after the ones it has made in the previous pipeline stage. Now remember that in the previous cycle the INS would have compared two items, written the smallest to the memory, while it would also have provided the greater one as a flowing item for

[15] Although this generically differs from the one the insert canceling will provide.

the current cycle and corresponding pipeline stage. This is the item that could violate the heap property and should thus be compared with the proper item of the next level. However, this also constitutes an item that could be rejected without affecting the validity of the heap property. This is so as any already written item[16] does not violate it. So this can constitute the item we will give to a just-issued DEL to be temporarily considered as the new root. The repositioning that will take place due to this DEL will eventually bring the heap to a valid state. Notice that with the mechanism of INS canceling the heap will not have the same form (i.e. same items at the same addresses) as it would have if we let the INSs reach the bottom of the tree. However this does not bring us against any drawback, as the needed heap property will hold at any time, ensuring the proper operation of the heap manager.

However, if no INS is currently active in the pipeline, then a situation arises where we must find and provide the "last" item of the heap in time. It may be residing in any address of any memory level. This situation is discussed thoroughly in the following subsection.

7.6.2 Locating and Being Able to Provide the Last Item In Time

The first thing we have to do is to find where the last item resides, or more correctly where it would reside in a steady-state situation (i.e. when all the operations in the pipeline have terminated so that no other repositionings happen). This information can be given by the counter, holding the number of items currently in the heap. The position of the most significant 1 in the binary representation of the counter actually points out which memory holds the last item, while the rest LS bits will point the place of the item in the specific memory. The simple thing to do would be to read this item and keep it to offer it at a forthcoming DEL. However, although this may be simple in a steady state, the situation is far more complicated at a dynamically changing heap state, with many operations being simultaneously active at different stages of the pipeline. First we have to deal with how we can read from the memory currently holding the last item (here on called last-memory in this section). The memories we use are dual-ported and we always use the first port for reads and the second for writes. We would like to read from the last-memory only when it is not given a write command, and thus being updated at that time, maybe affecting the location we aim to read. So we should use the second port for last-item reads, as we would anyway have to ensure the availability of this port. What is more, the first port should not be disturbed, as it must be let to be used for previous pipeline stages' reads, which do not whatsoever affect our last-item read.

[16] ignoring any other currently active operation in the pipeline.

Now suppose that we have a DEL just issued, that is seeking for the last item to "replace" the root. We firstly consider situation A in which a write is to be given at the forthcoming clock tick to the memory we would like to read. In this case we cannot give a read to the memory as our port (2^{nd} port) will be in use. However this write can be coming from either an INS, in which case we have an INS-canceling procedure, or from a DEL. In the second case we know that in the previous cycle the specific stage would have been idle (as a DEL always has a preccedding bubble), thus we would have been able to give a valid read at the second port at this previous cycle. This read will now be providing the last item in this case. However, if the DEL that will use the last memory for a write happens to write at the address holding the last item (i.e. aims to update the memory position we have previously read), we will have to forward this item to the DEL just issued.

In situation B the last memory is not to be given a write command for the next clock cycle. In this case we give a read command for the last item (at the 2^{nd} memory port as always). However the last item will be provided by the memory at the next clock cycle, too late for the DEL just issued, which claims it before the end of the current cycle. So this read only serves the purposes of a forthcoming situation A, and the returned last item will be kept in a register. Thus, we should already have the last item ready for the just issued DEL. This is again feasible. We just have to see what could have happened in the previous cycles. In the first case we had a DEL writing in the last memory. If it wrote the last item we will have forwarded this item and kept it in a register. If the DEL wrote to another address, then the valid last item would already be driven in the register, as we would have been able to give a read for the last item at the previous idle (being idle as it precedes a DEL). In the second case we had an INS writing in the last memory. In this case the valid last item would have been stored in a register through forwarding, as an INS at the last memory always writes the last item. Summing up, as the previous discussion shows, we are always able to provide in time the last item for a newly issued DEL.

Here, ending the discussion considering the last item delivering, we would like to mention a somewhat significant particularity. This concerns the oen signals of the memories we use. The DPR2 memories of Alcatel have all the inputs latched, except for the oen signals. In this way all the signals start to be taken under consideration after the clock tick that latches them, while an oen signal is monitored asynchronously. It actually drives a tri-state driver. This driver gives to the output the contents of a buffer which contains either the previous item read from the corresponding port, or the last item written (thus the item that participated in the last valid memory transaction). This could help us a lot when we want to read the last item while the last memory is given a write operation.

Instead of waiting for the next clock cycle to give a read operation to the last memory and wait for the next clock cycle to obtain the result, we could immediately activate the oen signal and have the results. However we used the method presented above, as it is more solid and what is more compliable to any other memory that could alternatively be used along with each pipeline stage. It also suits for the registers and register files used for the first levels instead of memory. The needing latching for the oen signal is provided by as for the specific memory modules we use. This makes them synchronous and as they are valid from the beginning of each cycle aid in shortening clock cycle[17].

7.6.3 A Stale Counter Problem for Delete's

As we have already stated, a DEL operation uses the counter information it has taken at issue time to decide when it should terminate (i.e. to find which entries of the ones read at each level contain valid data). However, this notion of which items remain valid, and should thus be considered for comparisons, is based on all the previously issued INSs being completed. In that way they will have placed a new item at the last level before the DEL reaches the last active level - just as the pipeline designates-. Only if this approach remains as it is, the counter value used by each DEL will give the proper results. However, each time an INS is canceled, it will not reach the last level. So the counter number used by a subsequent already issued DEL will no more correspond to the number of valid entries to be found in the tree. The solution for this is as follows; As the youngest INS is always the one to get canceled, the DELs to have invalid counter information will be the ones flowing in the pipeline from the issue level (level 1) down to the first INS (the one to be canceled). So a "-1" signal should be given to each of these DELs, so as to help them keep their counter number updated.

The same stale counter problem emerges even when no INS is active in the pipeline, so as to be canceled when a new DEL issues. This new DEL will then extract the last item. At such a situation it should notify all the DELs currently flowing in the pipeline. It will do so by sending the "-1" signal to all the levels of the pipeline, so that each active DEL receives it and updates its counter.

7.7 A Description of All the Possible Bypasses Involved

In this section we present all the operation sequences that cause the activation of a bypass path. We also give a small description of the associated data dependency. Each such sequence can be seen at figure 36 . Part (A) gives the sequences for a DEL (i.e. those

[17] If we used the alternative method the path would include data-selection + memory-write + memory-read through oen asynchronous activation, probably leading to a longer clock cycle.

ending with a DEL, on which we focus to describe the bypasses), while part (B) gives those for an INS. To give some explanation of how things are represented, when for example we give a DEL-INS sequence, we mean that a DEL is issued, and at the next cycle an INS follows. This means that in the pipeline the DEL flows at the level ahead of the one INS is flowing (i.e. is active, executes). When an INS is marked with a cross, this means that it gets canceled from a DEL issued later. Having these in mind and also referring to figure 31 and figure 32 , will help us understand how the data dependencies and the associated bypasses to overcome them come into appearance, as given in the following discussion.

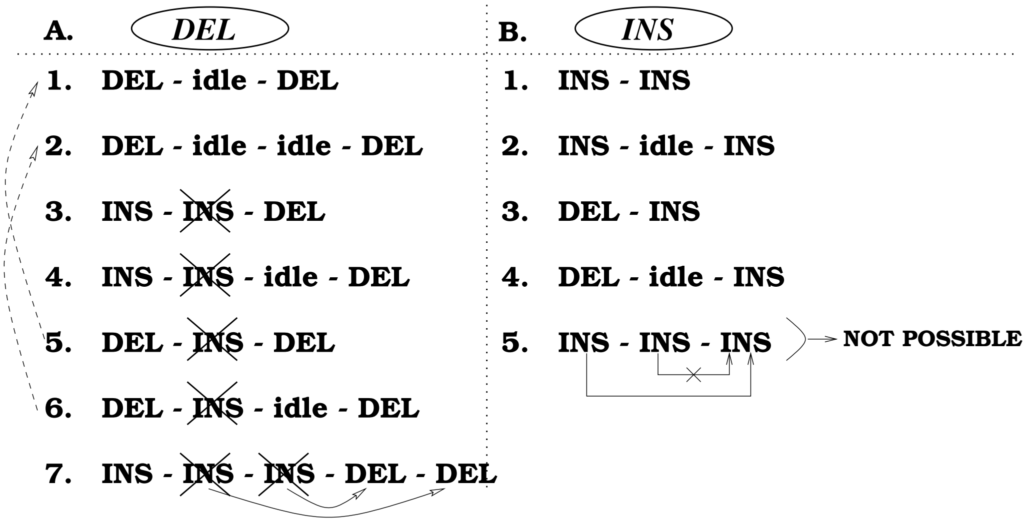

Figure 36: The Command Sequences Leading to the Need for a Bypass

Line (i) of figure 36 (A) concerns the sequence DEL-idle-DEL. This means that two DELs were issued, with the needed intervening idle cycle. The data hazards concerns the elements that the 2^{nd} DEL (DEL2) reads, while the 1^{st} DEL (DEL1)[18] has not yet written its result, possibly leading DEL2 obtain stale data. In this case, when DEL2 reaches stage 2 (figure 32) for example, the path named D3 will be activated to give the updated element, if indeed one of the items to be used by DEL2 is to be updated by DEL1. This check will continue accordingly as DEL2 flows through consecutive stages of the pipeline. Line (ii) is a similar case, but with two intervening idle cycles. In this case the path labeled D3 will possibly be activated once DEL2 occupies stage1, as at this time DEL1 will try to write L1 memory (the root). Line (iii) concerns a case where an INS proceeds a DEL in the pipeline. For this to be possible, another INS should have been initially issued, as the DEL will cancel the youngest INS. So the pipeline has a leading INS, a bubble and then the DEL. The case is now similar to that of line (i), but now the

[18]the sequence can thus be represented alternatively as DEL1-idle-DEL2

path labeled I3 (figure 32) will possibly be activated. Similarly line (iv) resembles the case of line (ii). Finally the cases of lines (v) & (vi), after the youngest INS is canceled, get transformed to cases (i) & (ii) respectively. One last case presented in line (vii) shows an example of how dependencies in the pipeline may change dynamically. In this example, when DEL1 is issued, it cancels INS3, and thus gets dependent on INS2. However, when DEL2 is issued, even if it was issued when DEL1 has flown deeper in the pipeline than presented in the figure, it cancels INS2. So from that point and on, DEL1 get dependent on INS1 and will have to check for data dependencies concerning the actions of INS1 (like the case of line (iv)), as INS2 no longer exists.

figure 36 (B) gives the command sequences that concern an INS and require bypassing. As we can see at a first glance, these command sequences are shorter, compared to those of part (A). This is so, as an INS that is active and occupies a stage of the pipeline, wants to use the elements of the same memory level (i.e. the needed read command is issued at that memory level). This was not the case for a DEL, which wants to read the elements of one memory level ahead. This makes an INS dependent only to operations occupying up to the next two stages of the pipeline, so the command sequences that may lead to bypassing reach up to length of 3, including the INS itself (while for a DEL it was 4). In line (i) of figure 36 (B) an INS immediately follows an INS. In this case we reach a moment where INS2 tries to use the data it has read, while INS1 has not yet written a new item, possibly updating the data to be used by INS2. So, when INS2 executes at stage2 (figure 31), the bypass path labeled I2 will be activated if needed. The situation continues accordingly to the next levels. In the command sequences of line (ii), an intervening bubble exists. Now INS2 reads the memory level at which INS1 tries to write. So now the data dependency appears sooner for INS2, and the path labeled I2 will (possibly) need to be activated when INS2 executes at stage1. With the belief that up to now things have started to be understood intuitively, we briefly mention that line (iii) activates the path labeled D2 (figure 31) when INS executes at stage2 and line (iv) activates the path D2 when INS executes at stage1. Finally line (v) presents a special case that someone could think of as possible. As we have mentioned up to now, an operation is concerned only for the data dependencies having to do with the actions of the youngest operation that is active and ahead in the pipeline. The latter will deal with the data dependencies for commands deeper in the pipeline. So we can understand that in a recursive manner the dependencies are solved, with each command dealing only with the dependencies due to just the command ahead of it. In line (v), as already mentioned, a special case is presented. Three successive INSs are issued. If at a certain point of the binary tree, each successive INS takes a turn different to its preceding INS, then INS1 and INS3 will have taken the same turn (e.g. left), while INS2 will have turned the other way (e.g. right). So from that point and on, INS3 will be data dependent on INS1 and not INS2, as was our prior claim. This would cause troubles, as it opposes to our general rule of how to handle data dependencies. Anyhow, this case cannot be encountered (but nevertheless gave us the incentive to present an aspect of the generic way the dependencies are dealt with). If we think of how the tree is formed, and with the very helpful property of it always remaining fully balanced, its leaves will get filled up from left to right, making it impossible to have the case of the turns described above. So our general rule of how data dependencies are resolved remains sound.

7.8 Memories: Sizes, Speed and Addressing

In this section we will discuss about the memory configuration used as well as the associated memory performance and area. The binary tree manipulated by our design has a number of levels, each of different size. Each level holds twice the number of elements than its previous one. As different operations occupy different levels of the tree, many write and read operations need to be issued during a single clock cycle. However, as discussed earlier, we have managed to spread them to different levels. Therefore we would like to have each level of the tree implemented as a distinct memory module. So we will have as many memory modules as the number of the tree levels. This is shown in figure 37 for a four-level tree structure (containing up to 16 entries). As the first levels are too small to require a memory module to be used, we use registers to hold the first four. The rest are held in distinct memory modules. We need each memory location to be 32 bits wide, to hold the id and number composing an element. So we can at first aim for a 32-bit wide memory. However as for delete operations we want four elements to be read in parallel (i.e. in the same cc), we would like the memory to be at least $4 \times 32 = 128$ bits wide. As the four elements that need to be read at any level are in consecutive positions and properly aligned, 128-bit wide memories having the consecutive elements of each level stored with the same order would suit nicely for our purpose. What is more, when we write an element while executing either an insert or a delete, we want to write just a single 32-bit position. For this to be feasible (and without having to always read and remember the other 3 elements comprising a 128-bit aggregate) we partition each memory to 4 32-bit wide memory modules. In this way we can write to just one memory module (having each address pointing to a single element's position) while we can also be able to read four consecutive elements, as each is stored at the same address for all four memory modules (by providing them that same address and enabling the oe/cs signals).

The addressing of the memories has thus up to here been briefly overviewed. We compute the address (actually the tree position number) of the needed element(s) as we

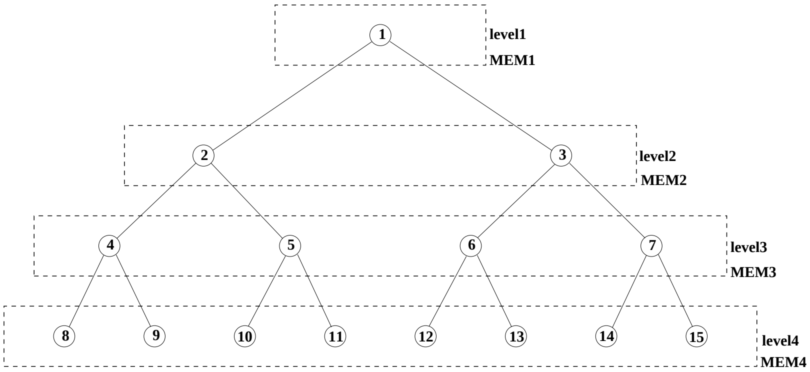

Figure 37: Each tree level stored in a distinct memory module

go down the levels. For reads we give this number as a read address to all four memories without the one MS and the two LS bits, while the latter two bits are used to prepare the oe/cs signals. We act accordingly for write's, giving the same address and also the same element input to all four memory modules, letting the we/cs signals to point the truly active module. As we often want a read and a write to (generically) different addresses to be executed simultaneously at the same tree level and thus same memory module, dual ported memories are required. One last intricacy is that in some cases we want to read from the same address of the same memory that we also want to write in parallel. However when this happen we actually do not really need the element tried to be read, as this will be bypassed. The value to be written will be forwarded and used instead of the value read, as this need for conflicting read/write signifies stale data at that address. This will have already been observed by the relevant bypass logic and the associated forwarding path will have been activated. So we only need the new element to be stored correctly and we do not care if the memory also correctly provides the element formerly stored there. Dual ported memories provide such a functionality and are thus suitable.

7.9 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 4.3.1). 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. Patterns are generated by a parametrized test pattern generator written in Perl. Various parameters can be given, such as the maximum occupancy count to be reached, the kind of sequences to be given (such as random, or continous INSs followed by continous DELs to succeessively activate the insert abortion mechanism, etc), or the minimum number of entries that will be allowed to be on the tree at any time (keeping it deep enough for complicated mechanisms to be activated, or allow the heap to get empty to check its behaviour at such a case). The VC number of an element inserted is given from an empty pool of VSs. This pool gets updated with any operation issued. To know which VC will get freed up by a delete, the generator has knowledge of the way the heap operates (with the help of the plain perl model). What is more, the priorities are given in a way that satisfies the needs of the comparison mechanism presented in section 7.3 , thus the generator has knowledge of the wrap-around paradigm. Finally perl scripts are used to verify proper operation of the verilog design, compared to the verification models.

7.10 Improvements

Carrying on our work, we tried to find ways to achieve a higher clock rate, after having the results from the synthesis procedure. The most time consuming paths are two: (i) the path that finds the proper item from a canceled insert operation. and, (ii) the path that brings the last item from the memory, or by a bypass from a memory write access. Both of these paths are used to provide the new root for a newly issued delete operation. If these two paths were left out, then the resulting clock cycle is close to 11 ns (actually a bit less). So, to shorten the cycle of the overall design, we should try to break these two paths and perform the needed functionality in two consecutive cycles. This seems feasible, and the way to achieve this is as follows.

First of all, in order to find the needed extra cycle (for the functionality to be broken apart), we should add a stage to the pipeline presented earlier. This can be named stage 0, and it will precede stage 1. So, any newly issuing operation will enter stage 0, and in the next clock cycle it will move on to stage 1. From thereon, operation will continue just as was previously described. The difference now is that when a delete issues, an added cycle was already used to perform some preparatory processing (the first part of the two to be created by the splitting of the paths we will describe), before it moves to stage 1 where the new root should be finally decided and be ready to be passed to stage 2. In the case of an insert, this added cycle will be useless, but will do no harm.

Let us now see how the insert canceling can be broken into two parts. The first part is to locate the youngest currently active insert operation using a priority enforcer with one input per heap level. The second part concerns the multiplexing of all the possible sources for a canceled item, to select the actually canceled one. This is a num_of_stages-to-1 multiplexor. Also a small multiplexor to select a canceled item or another source is included. This partitioning seems to fit well, as far as timing is concerned. The whole path for the insert canceling takes about 15.5 ns (it is the second slower path), while the aforementioned first part constitute for about 5.5 to 6.0 ns. So the resulting partitioning will probably conform to the 11 ns limitation, imposed by the basic pipeline stage's delay. The only problem that appears with the partitioned mechanism is that the insert we have selected in the first cycle (stage 0) will have decended by one heap level when the inputs will need to be taken for the multiplexing of stage 1. To overcome this, in the first cycle we use the signals shifted right by one the register holding all the insert-valid signals, before the priority enforcer uses it. In this way we emulate a future pipeline snapshot (one cycle ahead), producing the correct signals.

The other worst path is that of bringing the last item. The decision selection of the last item includes memory access, and then multiplexing to select the appropriate memory output. Also, in parallel, write addresses are compared with the last item address, in order to determine the need for forwarding of the item that is being written, and multiplexing again follows. The notification to be given to some of the active delete operations in the pipeline, in order to update their counters is also produced. So, the first of the two cycles (stage 0) can be used for the memory access. Thus we read the last item from the memory where it resides. The multiplexing will follow in the second cycle. However, this multiplexing is again decomposed into two separate stages. One includes the selection of one stage's memory (the stage that contains the last item), and the other the selection of the proper item, as each stage's memory produces four possible items (4-entry wide memories). This second selection (4-to-1 mux) can be included in the first cycle (stage 0) in order to have only the other num_of_stages-to-1 mux in the second cycle (stage 1). The selected items from stage 0 (one for each stage) are kept into registers, which will constitute the inputs of the mux in stage 1. For the case a write access in performed to the last item, the address comparisons can also be made in the first cycle, and if the write address matches the last-item address, then this result is the one to enter the corresponding stage's register (to be used for the num_of_stages-to-1 mux input). Another functionality that can be included in stage 0 is the process of notifying the other delete operations to update their item counters. This will lead to this updating being performed early enough, so that no delete will try to update the last item once the new delete have moved on to stage 1, and thus have already decided the inputs for the large mux. As the functionality included in stage 0 accounts for about 7 ns (out of the worst path's 17 ns), the resulting design will again probably conform to the 11 ns limitation.

7.11 Resulting Cost & Performance

As we have mentioned before, our implementation constits of 32-bit entries, and 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[19] plus 500 bits of flip-flops/registers. As we mentioned, the first level is the 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 38 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[20]; the limiting factor that determines this clock frequency is the cycle time of the largest memory block[21]. This clock frequency is for a low-power, low-

[19] simple 2-input NAND/NOR gates

[20] 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.

[21] Based on the improved pipeline of section 7.10.

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

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.

Conclusions

We have presented an overview of advanced scheduling algorithms. As we saw, finding the minimum among a large number of values was a common computational need. The heap data structure was then presented as a promising basis to work through this problem. The modifications needed to be applied to the traditional heap algorithms were then presented. Continuing with our work, we have implemented a pipelined heap manager, thus demonstrating the feasibility of large priority queues with throughput rates in excess of 100 million operations per second at clock speeds of a few hundred MHz. The cost of these heap managers is the (unavoidable) memory that holds the priority values and the flow ID's, plus a dozen or so pipeline stages of a modest complexity. This compares quite favorably to calendar queues –the alternative priority queue implementation– with their increased memory cost and their complexity for handling hash collisions. We have presented the way the heap manager operates in great detail, giving solutions to the many problems arising when pipelining was introduced. We furthermore gave a set of alternatives that trade off performance and cost. These signify the reason that the current implementation was carried out. It can furthermore comprise the basis for less costly solutions, if someone is interested in implementing such. What is more, many of the solutions given to various problems can be used distinctly, if analogous problems appear at

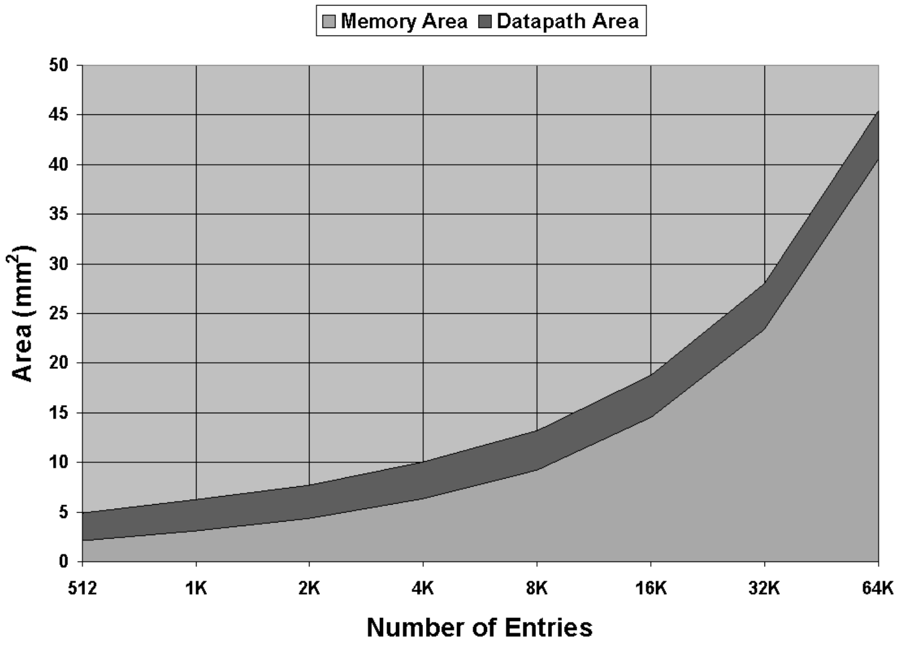

Figure 38: Approximate memory and datapath area in a 0.18-micron CMOS ASIC technology, for varying number of entries

other, maybe unrelated, implementations.

The feasibility of priority queues with many thousands of entries in the 100 Mops range and above has important implications for high speed networks and the future internet. 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 are demonstrating the feasibility of these algorithms, at reasonable cost, at OC-192 (10 Gbps) and higher rates.

Appendices

A How the Performance for the Memory Alternatives Is Figured Out

In section 6 where we present alternatives for the memory configuration of the heap manager, the resulting performances were evaluated after cautious analysis. Here we give a brief overview of how the resulting performances were figured out.

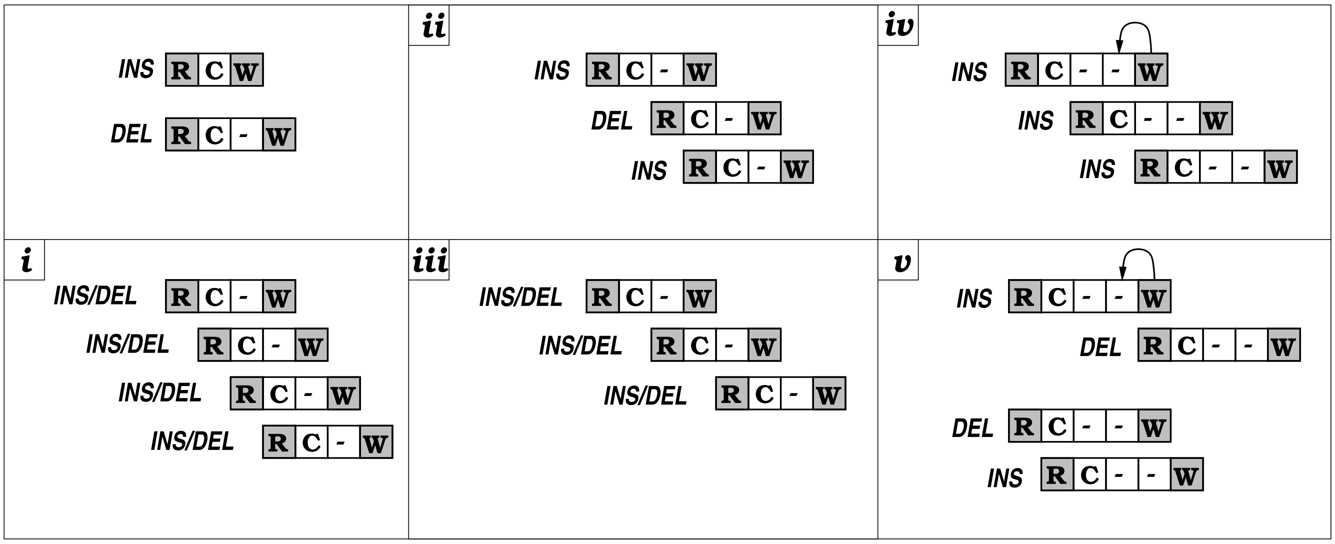

Figure 39: The Scheduling of Commands to Achieve the Desired Performance With Various Memory Alternatives

Each operation executes a Read (R), then a Compare (C) and finally a Write (W) command. An INS executes them as seen at the top-left part of figure 39 , at the sequence labels "INS". The three commands (R-C-W) are executed at consecutive cycles. DEL on the other side executes the sequence labeled "DEL" for a specific level of the tree (R-C-noop-W). This is so as at the 3^{rd} cycle a DEL writes at the previous level's memory, and – only when the DEL traverses the tree at the speed of 1 level/cc – it will write the memory of the current cycle at the 4^{th} cycle. Continuing with figure 39 , case (i) concerns case (i) of *Table 1* . We can see how consecutive operations can be issued and execute at a specific level of the tree. The "W" for an INS has been shifted one cycle later, resembling the execution patter of a DEL, in order to benefit from homogeneity, which relaxes the operation scheduling from write-back structural hazards. In part (ii) of figure 39 we see that the same execution pattern is helpful for the case of line (ii), *Table 1* , with a DEL requiring an intervening idle cycle. As we are concerned about how the pipeline works with the commands already flowing through the tree, a DEL is always preceded with an idle cycle, as it is considered to already have canceled the youngest INS (if any).

Notice that in both cases (i) & (ii) a "R" of one operation can be executed at the same cycle with a "W" of another operation, as the memories are dual-ported. This is not the case for the following memory configurations that use single-ported memories. Part (iii) of figure 39 concerns the case of lines (iii) & (iv) of *Table 1* . The situation is intuitively understood, so we give no further explanations. However part (iv) (for table line (v)) deserves some extra attention. To have the pipeline serve our needs, by being homogeneous for simplicity while also avoiding structural hazards, we make each command sequence last 5 cycles, with 3 intervening cycles between a Read (R) and the corresponding Write-Back (W). However this sequence should be of variable length. This is so, as for example in the case of figure 39 (iv), where three consecutive INSs flow through the pipeline, the "W" of the 1^{st} conflicts with the "R" of the 3^{rd}. So the former should be moved one cycle before to avoid both the hazard and an added idle cycle before the 3^{rd} INS issues. And this does not complete the analysis of this case. When an INS is issued, we can think of it as it leaving a single idle cycle, while a DEL leaves two, which conforms to the performance evaluation shown in *Table 1* (v). However due to data dependencies when we move from one command to a different one, the situation is somewhat different, and is dealt with the timing given in figure 39 (v). However the performance evaluation shown in *Table 1* holds, as we can still think of a DEL spending 3 cycles and an INS two. This is so as a DEL issued after an INS will spend 4 cycles, but when the sequence of DELs (probably composed of just that single DEL) and we reach the following INS, the extra cycle will be 'repaid'. So the earned cycle at a DEL-INS sequence could be though of not actually giving credit to an INS but actually returning lost credit to a DEL.

We give no further analysis for the next line of *Table 1* , as similar pipeline sequences (but of greater length) can be figured out, and with the accompanying help of the descriptions at section 6 , to achieve the indicated performance.

B Various Code Information

Verilog Code	3200 lines
Automated Verilog Code	3100 lines
Total Compiled Code (without memories)	6300 lines
Perl Code	450 lines
C Code	550 lines
Test Pattern Sizes	up to 100K op's
Scripts for Synopsys	250 lines

References

[Rexford96] J.L. Rexford, A.G. Greenberg and F.G. Bonomi, *"Hardware-Efficient Fair Queueing Architectures for High Speed Networks"*, INFOCOM '96, pp 638-646

[Tarjan83] Robert Endre Tarjan, *"Data Structures and Network Algorithms"*, (CBMS-NSF Regional Conference Series in Applied Mathematics), 1983, ISBN 0-89871-187-8.

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

[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.

[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. .ḧttp://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.