HARDWARE SUPPORT FOR QUALITY-OF-SERVICE GUARANTEES IN PACKET SWITCHED NETWORKS

by

Sung-Whan Moon

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2001

ACKNOWLEDGEMENTS

I would like to thank everyone who supported and assisted me during the course of my graduate studies at the University of Michigan. First and foremost, I would like to thank my advisor, Professor Kang G. Shin for always looking out for me both in and out of the lab. Not only did he support me for so long, but he gave me the freedom to pursue any research interest I had. I would like to thank Professors Richard Brown, Trevor Mudge, and Steven Reinhardt for serving on my thesis committee and for their advice and support.

Thanks also go out to my parents, family, friends, and office mates for their support throughout the course of my graduate studies.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Recent advances in network technology has brought about substantial increases in bandwidth along with wide-spread Internet access through high-speed connections beyond the corporate and academic communities and into homes. This has resulted in the introduction of a large number of applications with a wide range of quality-of-service (QoS) requirements [12]. Representative of these new applications are those with real-time traffic, such as video and audio. This real-time communication [2] [75] requires guarantees such as bounded end-to-end delay, bounded cell-loss rates, and guaranteed bandwidth from the network. Today's packet-switched networks can employ a variety of methods to provide the QoS guarantees for present and future applications.

As a packet traverses the network towards its destinations, contention for resources at each node within the network will result in queueing of some packets while others receive service. The goal of the various QoS mechanisms is to determine how to allocate resources to the queued packets such that the QoS requirements are satisfied for the corresponding applications. Assuming packets can be efficiently classified (classification determines the application, flow, or connection to which the packet belongs), mechanisms such as packet marking (which determines the packets to be dropped with a certain probability in case of buffer overflow), traffic shaping, and link scheduling can be used to deliver QoS.

The goal of traffic shaping is to monitor and control connections so that they abide by their connection traffic parameters. Because schedulers can distort the traffic envelope of a connection, packets need to be re-shaped if they are to receive their desired QoS and not disrupt other connections down stream. Link schedulers multiplex among packets from different connections onto a single link for transmission. The order in which queued packets are served is determined by the link-scheduling algorithm. When a packet enters a node, it is first placed into the shaper. If the shaper determines that the packet is abiding by its traffic parameters, the packet is considered eligible for transmission and moved into the scheduler queue, which will eventually schedule it for transmission. Otherwise, the packet will remain in the shaper until it becomes eligible.

Shaping and scheduling algorithms differ in the QoS guarantees they can provide to individual connections (e.g., end-to-end delay bounds, delay jitter, distribution of unused bandwidth). However, realizing the benefits of the algorithm requires an efficient implementation which can process a large number of packets on a high-speed link. Due to the very high link transmission rates and very small amount of time available to process each packet, a hardware implementation of the shaper and scheduler is needed to transmit packets at link speeds. For example, a 64 byte (1,500 byte) packet on a 1 Gbps link allows for 5,120 ns (12,000 ns) between successive packet transmissions. In a worst-case scenario the scheduler must determine the packet to transmit every 5,120 ns (12,000 ns), while the shaper is moving one or more eligible packets into the scheduler within the same 5,120 ns (12,000 ns). At the same time, the shaper must also process one or more newly-arrived packets. Software solutions are typically not fast enough to keep up with the packet transmission rate due to the associated overheads (i.e., in requesting service from the processor, sending and retrieving data from the processor). On the other hand, a hardware solution can operate close to the operating speeds of the link.

This dissertation deals with the issues involved in the design and implementation of hardware architectures for traffic shaping and link scheduling. In particular, we focus on the key basic components of any traffic shaping and link scheduling implementation which must operate on every packet. Migrating these simple and highly repetitive tasks to a hardware implementation can significantly improve performance. However, this performance must not limit the overall flexibility of the implementation, as the specific traffic shaping and link scheduling algorithms implemented can change in response to the current needs of the network, user, or network designer. This thesis examines several hardware architectures, with an emphasis on the following three attributes:

Performance: An effective implementation requires that it be able to process a very large number of packets within a very small amount of time.

Scalability: Changing the design parameters for an implementation should not require changes to the architecture. In other words, the architecture must scale easily to different parameters. Also, scaling to larger design parameters should not degrade the performance of the implementation.

Flexibility: The architecture must be capable of supporting a wide range of traffic shaping and link scheduling algorithms. A hardware implementation does not necessary mean a fixed solution. A flexible architecture allows a network designer to change the algorithms to adapt to different demands.

By incorporating these features, we propose architectures which combine the performance of a hardware implementation with the flexibility of a software implementation capable of implementing any mix of traffic shaping and link scheduling algorithm.

1.1 Application Domain

This research concentrates on traffic shaping and link scheduling in a packet-switched network. Each node within the network provides a switching function by forwarding incoming packets to their correct outgoing links. For the traffic shaper and link scheduler architectures discussed in this paper, we consider a common router/switch architecture, as shown in Figure 1.1. Throughout this thesis, we will use the words "switch" and "router" interchangeably. The router [40] is characterized by a packet processing unit at each incoming and outgoing link, a shared buffer space and output queueing [36]. The input link processing units provide per-packet services such as packet classification and route decision. Each output link incorporates a traffic shaper and link scheduler. Although other memory configurations are possible [73], output buffering offers better performance than input buffering while a shared buffer configuration has better memory utilization. When a packet enters a node, the packet classifier determines the connection to which the packet belongs (we will refer to this as the flow/connection id/number) [35] [42] [84], while a route decision is made to determine which output link to forward the packet [34] [37] [55] [83]. At the same

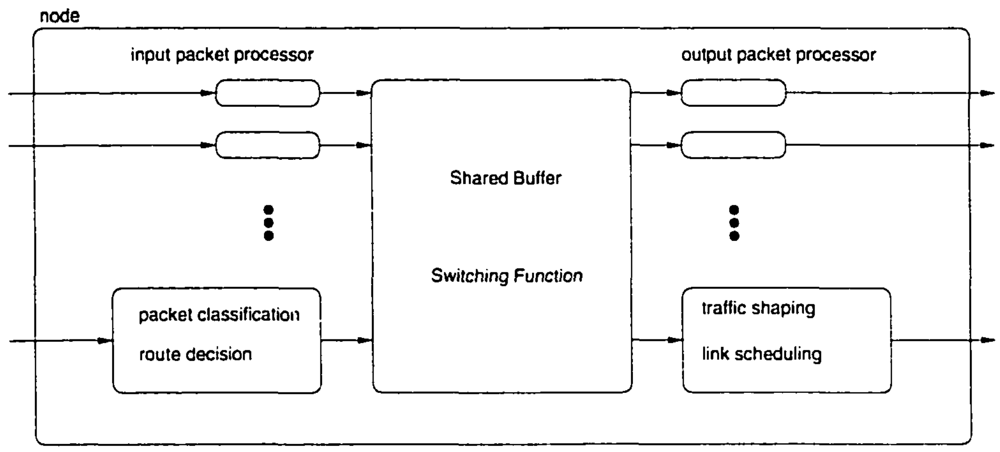


Figure 1.1: Simplified block diagram of a single node with a shared buffer and per-link packet processing.

time the packet is stored in the buffer and an address (pointer to memory) and length of the packet are returned. A tag consisting of the flow id, a pointer to the packet, and the packet's length, is created for each packet and forwarded to the output link specified by the route decision.

At the output packet processor, the tag is processed by the traffic shaper and placed into a shaper queue. It will remain there until the shaper decides the flow is conforming to its traffic parameters. When the packet corresponding to the tag is eligible for transmission, the shaper calculates its priority and releases it to the scheduler by writing the tag into the scheduler queue. The job of the scheduler is to determine the next packet to transmit when the link becomes idle, with the packet chosen among those in the scheduler queue based on the priorities of the packets. The goal of this thesis is to identify the mechanisms required in a traffic shaper and link scheduler implementation, study the issues and problems that need to addressed for an effective implementation, and propose new architectures for implementing these mechanisms.

1.2 Contributions of the Dissertation

This dissertation makes several research contributions related to the design and implementation of traffic shaping and link scheduling algorithms.

Priority queue architectures: We highlight the challenges of building scalable priority queue architectures, and propose two new architectures which scale to the large number of packets (N) and the large number of priority levels (P) necessary in supporting modern high-speed networks and a wide range of algorithms.

Traffic shaper and link scheduler architectures: We highlight the challenges in building a scalable, flexible, and effective integrated traffic shaper and link scheduler architecture. We propose two new architectures which use novel techniques to solve key problems in the design of an integrated mechanism.

Network interface architecture: We motivate the need for providing traffic shaping and link scheduling services on and end-host server, and present a network interface architecture with a dedicated shaper-scheduler mechanism.

A shaper-scheduler processing (SSP) engine: We analyze a wide range of traffic shaping and link scheduling algorithms and extract a common framework. Based on this study, we develop a small set of instructions and present a microcontroller-based shaper-scheduler processing engine which provides the flexibility of a software implementation with the high performance of a hardware solution.

A hardware/software co-design and co-simulation tool: Finally, we present a simple C-based event-based simulator which can handle both hardware and software components of a design without the need for expensive communication mechanisms, such as inter-process communications and exchange of control and data information, between the hardware and software components. This tool allows for the quick implementation and evaluation through simulation at a fraction of the time required by traditional hardware simulators such as NCVerilog.

1.3 Outline of the Dissertation

The rest of this dissertation is organized as follows:

Chapter 2 includes a detailed comparison of four hardware priority queue architectures from the current literature. Based on these comparison results, two new architectures are presented. Using the Verilog hardware description language and the Epoch silicon compiler, we designed and simulated these two new architectures, as well as the four existing architectures. The simulation experiments compare the designs across a range of priority queue sizes and performance metrics, including enqueue/dequeue speed, chip area, and number of transistors.

Chapter 3 begins with a detailed study of several well-known traffic shaping and link scheduling algorithms, and we detail a set of key mechanisms that any implementation will require. We compare several existing shaper-scheduler architectures, and present two new architectures. An evaluation of each architecture focuses on its scalability in terms of flexibility and implementation issues.

Chapter 4 presents an end-host server network interface architecture with a dedicated shaper-scheduler mechanism. A study of alternative solutions provides the motivation for dedicated support on the network interface. Simulation experiments show how an audio/video streaming server can take advantage of our architecture to provide improved performance (as seen by end clients) for a large number of clients.

Chapter 5 breaks down traffic shaping and link scheduling algorithms to obtain the basic set of instructions and data structures necessary to program them. Based on this study, a shaper-scheduler processing engine and a *complete* shaper-scheduler architecture is presented. Several example implementations using different mixes of algorithms are shown to evaluate the overall architecture's efficacy and performance.

Chapter 6 describes a hardware-software codesign and co-simulation tool which allows a designer to model and simulate a mix of high-level and detailed descriptions of both software and hardware components of a design on the same simulation platform.

Chapter 7 concludes this dissertation with a brief summary of our contributions and examines possible future directions for this research.

CHAPTER 2

HARDWARE PRIORITY QUEUE ARCHITECTURES

2.1 Introduction

The simplest link-scheduling algorithm is first-in-first-out (FIFO). The problem with this approach is that it is characterized by poor utilization of resources and poor performance. In particular, a FIFO scheduler cannot admit many new connections, especially when the link services connections with a wide range of traffic parameters and QoS requirements. Other link-scheduling algorithms achieve better performance by assigning a priority number to connections or individual packets. This priority field can represent a traffic class, a deadline, a virtual finishing time, or a sequence number, depending on the link-scheduling algorithm. Once the priority number is determined, a priority queue ranks packets based on the priority assignment. The net effect of the link-scheduling algorithm and the priority queue is to interleave the packet transmission from the various connections such that each connection's QoS requirements are satisfied.

The priority queue is essential in implementing the link-scheduling algorithm. Due to the high-speed at which the networks operate, a hardware priority queue [60] is needed to transmit packets at link rates. For example, in a 155 Mbps (2.5 Gbps) Asynchronous Transfer Mode (ATM) network, an ATM cell can be transmitted every 2.7 μsecs (0.17 μsecs). In a worst-case scenario the priority queue must determine the next highest priority cell (dequeue operation) every 2.7 μsecs (0.17 μsecs), while being able to accept new cells (enqueue operation) from all incoming links within the same 2.7 μsecs (0.17 μsecs). Software solutions, which are logarithmic in time complexity, are typically not fast enough to keep up with the packet transmission rate due to the associated overhead (i.e., in requesting service from the processor, sending and retrieving data from the processor). On the other hand, a hardware solution can operate close to the operating speeds of the link. Also, a hardware solution can overlap enqueue and dequeue operations with packet transmission to avoid wasting link bandwidth.

As we mentioned in the previous chapter, after a packet is stored in the shared buffer, a tag is used within the scheduler to refer to that packet. Within the context of a priority queue, this tag consists of a valid/invalid bit, an address ($\log_2 N$ bits), and a priority ($\log_2 P$ bits). Here N is the total storage capacity of the shared buffer measured in number of packets, while P is the number of priority levels supported in the link-scheduling algorithm. The priority queue is responsible for storing the entries and calculating the highest-priority entry when the output link is ready to transmit another packet. So, regardless of internal architecture the priority queue must provide for the storage of packet tags, initialization (clear contents of the priority queue), enqueue of new tags, and dequeue of the highest priority tag.

Since a switch's buffer size (N) and the number of priority levels (P) needed by the link scheduler can be both very large, the priority queue must be easily scalable to these two parameters. That is, the total entry capacity of the priority queue must match the total packet capacity of the shared buffer, and it must support a large number of priority levels. At the same time, the priority queue's performance must not fall behind link rates as it is scaled to N and P. If this were to happen, then a link will remain idle even though there are packets to be transmitted. Finally, the priority queue design should scale well with the number of output ports in the shared-memory switch, instead of requiring completely separate logic for each outgoing link, as in existing architectures.

We present two new priority queue architectures which were designed to minimize the effects of scaling (with respect to N and P). The first new architecture reduces and controls the performance loss due to increasing the queue capacity without adding a large amount of extra hardware. This was done by combining the salient features of two existing priority queue architectures, the shift register [13][15][74] and systolic array [44][45]. We then extend this architecture to service multiple links instead of just one. Both of the new architectures perform well enough to support very high-speed links, and both provide constant-time (in terms of number of clock cycles) enqueue and dequeue operations. But before describing our new architectures, we first describe four priority queue architectures –

binary tree, FIFO, shift register, systolic array – from the current literature in Section 2.2. A brief description of each architecture and operation is given, followed by a discussion on limitations to their scalability. Two new architectures are then proposed and evaluated in Section 2.3 (hybrid systolic/shift) and in Section 2.4 (multiple link). Each of these two sections also gives a detailed explanation of the new architecture's operations. Section 2.5 presents the results of some implementations of the various priority queues for several switch parameters. The implementations were done using the Verilog hardware description language and the Epoch silicon compiler for several combinations of P (up to 256) and N (up to 1024). These results show limitations of the existing architectures when scaled to large N and P, and are compared to implementations of the new architectures. Section 2.6 concludes with a summary of the chapter.

2.2 Priority Queue Architectures

This section presents four priority queue (PQ) architectures from the current literature. The FIFO and the binary tree architectures are the more intuitive ones. However, these two architectures do not scale well with increasing N and P. The shift register and the systolic array architectures take a different approach and scale much better than the FIFO and binary tree. The following subsections describe each of these architectures and discuss the effects of scaling on architectural complexity and implementation.

2.2.1 Binary Tree of Comparators

A binary tree comparator architecture [56][59], shown in Figure 2.1, consists of an N-entry storage block and a comparator tree of depth $\log_2 N$, whose output is the highest-priority entry among those in storage. A feedback mechanism is used to remove the output of the tree from storage. An advantage of this architecture is that the comparator tree logic can be shared among several storage blocks, reducing hardware costs. A disadvantage is that FIFO ordering is not maintained among entries with the same priority. Such FIFO ordering is important when applications assume that packets at the same priority level will arrive in the same order in which they were sent. Increasing N results in more leaf nodes (i.e., compar-

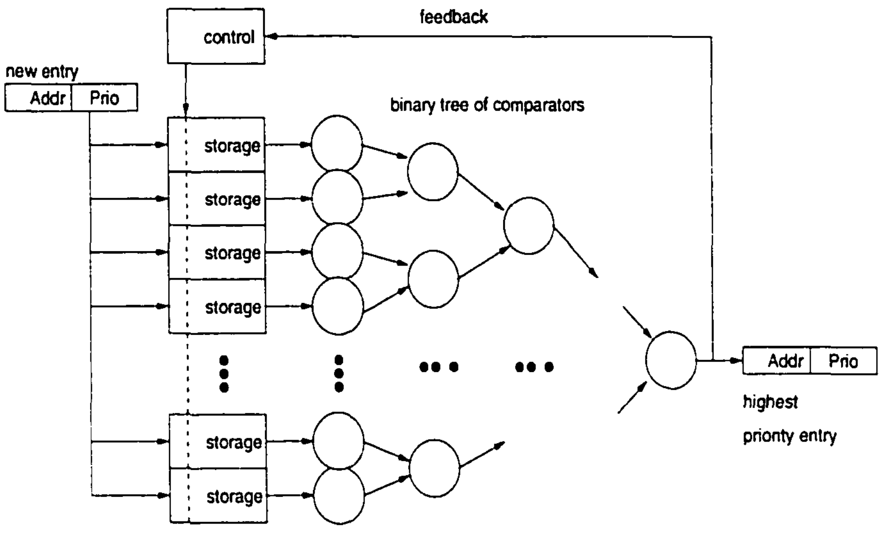


Figure 2.1: Binary tree of comparators priority queue.

ators) being added to the tree and increasing the capacity of the storage block. Problems with such scaling include bus loading problems with distributing the new entry to each storage element in the storage block, and increased dequeue time resulting from an increase in depth ($\log_2 N$) of the comparator tree. A possible solution to the increased dequeue time is to pipeline the comparator tree operation to reduce the clock period and increase performance; this can be useful if the comparator tree is shared among several outgoing links [59]. Another solution is to initiate the comparator tree only after dequeue operations and use extra logic to handle entries that arrive during and in between dequeue operations. This takes advantage of the fact that packet transmission time is longer than the comparator tree operation [56].

2.2.2 FIFO Priority

Like the bucket sorting algorithm, the FIFO PQ architecture [11][13], shown in Figure 2.2, inserts entries into one of the P FIFOs based on the entry's priority. Since each FIFO corresponds to a particular priority level, the queue does not need to store a priority field with each entry. During a dequeue operation, a priority encoder scans the head of the FIFOs in decreasing priority order and removes an entry from the first non-empty FIFO. Increasing

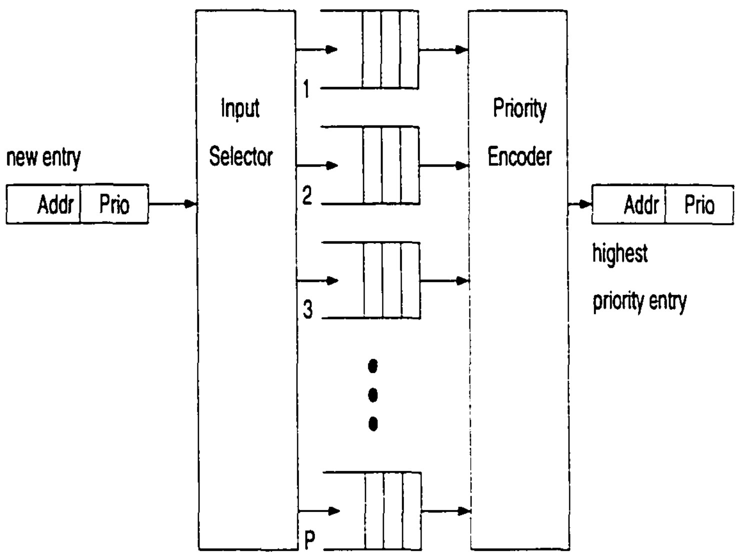


Figure 2.2: FIFO priority queue.

P requires adding more FIFOs, which results in added hardware costs and increased complexity of the priority encoder. Using logically linked lists [13][87] instead of physical FIFOs can reduce hardware costs. But this approach still suffers from the complexity problem of the priority encoder for large P.

2.2.3 Shift Register

The shift register PQ [13][15][74], as shown in Figure 2.3, consists of an array of blocks that store the entries in sorted order. Each block stores a single entry and communicates with the blocks immediately to its right and left. Higher-priority entries are stored to the right of lower-priority entries, with the 0th block containing the current highest-priority entry. On an enqueue operation, the new entry is broadcast to all the blocks via the new_entry_bus. Each block makes a local decision as to what action to take, with only one of the blocks latching the new entry. The others will either keep their current entry or latch the right neighbor's entry. The net effect is to have the new entry force all entries with lower priority to shift one block to the left, while the new entry places itself to the left of the entries with higher and equal priority. The lowest priority entry is discarded during an

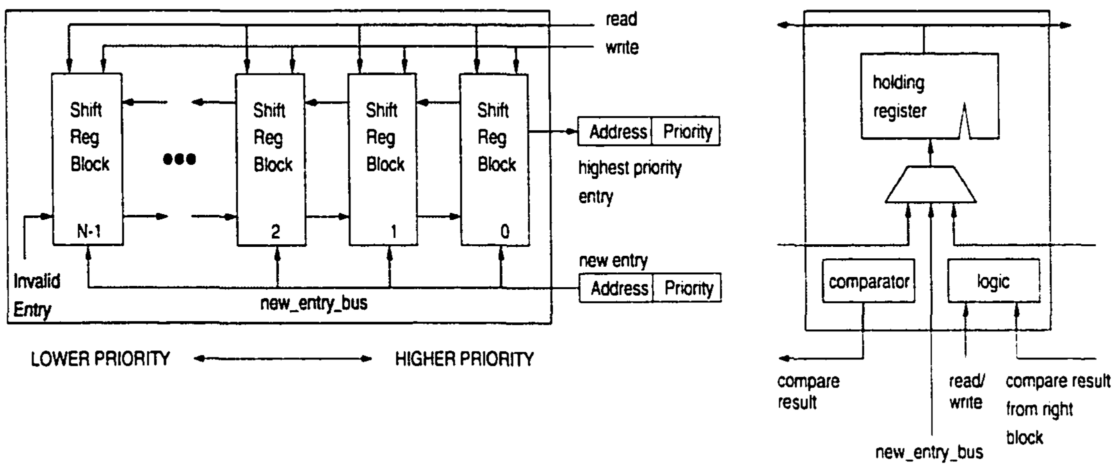


Figure 2.3: Shift register priority queue and shift register block

enqueue if the queue is full. A dequeue operation in the shift register simply reads the 0th block's entry while all other entries shift one block to the right.

As shown in Figure 2.3, each block consists of a holding register which stores the entry, a comparator which compares the priorities of the entry on the new_entry_bus and the holding register, a multiplexor (to choose from the left, right or new entry) and decision logic [13][15]. Since each block stores one entry, the queue's capacity can be increased by adding more blocks to the existing queue. Because each block makes decisions based on just local information, increasing queue capacity does not require modifications to the block's decision logic nor any central control logic for the queue. This makes scaling for large N very simple. As P increases, additional bits are added to the priority field in the entry's tag. This simply requires modifying each block's storage requirement and its comparator.

Unfortunately, implementation problems limit the scalability of this architecture. As seen in Figure 2.3, before any decision can be made by each block during an enqueue operation, the new entry must be present at the inputs of all the blocks. At the VLSI level, the new_entry_bus must be routed to the inputs of all the blocks in the array. As we saw with the binary tree architecture, this creates a bus loading problem, which adds to the hardware costs (buffers), and decreases the maximum operating speed of the queue. Thus, the shift register architecture's scalability with respect to N is limited by performance, not by archi-

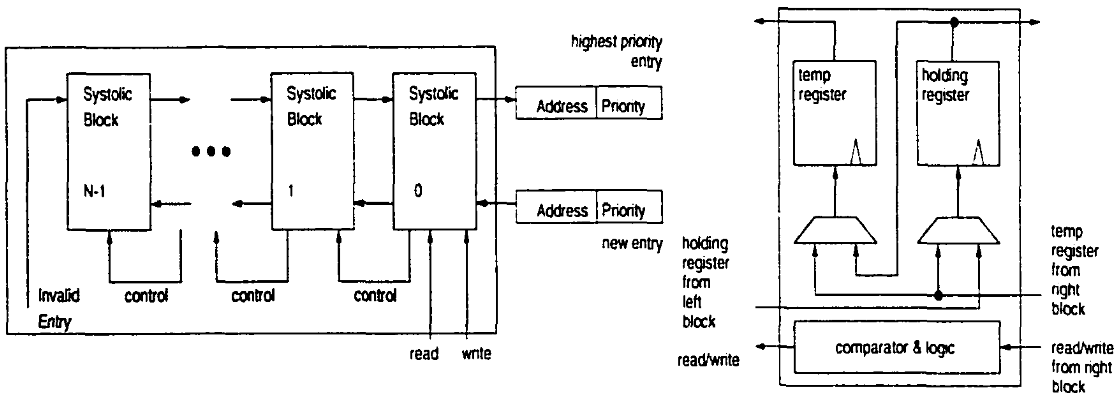


Figure 2.4: Systolic array priority queue and systolic array block

tectural complexity. Performance also decreases as P increases due to the added delay in the comparator logic. This is because the comparator's time complexity grows linearly (for a serial comparator) with the number of bits in the priority field.

2.2.4 Systolic Array

The systolic array PQ [44][45] is shown in Figure 2.4. Similar to the shift register architecture, the systolic array architecture consists of an array of identical blocks, with each block holding a single entry. On an enqueue operation, only the 0th block compares priorities of its entry and that of the new entry. On the next cycle the lower-priority entry is inserted into the left neighbor's block which repeats the same process of comparing and sending the lower-priority entry to the next block. So the systolic array does not become fully sorted until several cycles after the new insertion. Despite this feature, both insertion and removal still remain constant-time operations from the outgoing link's point of view. Because each block passes the lower-priority entry to the next block, the 0th block always holds the highest-priority entry in the queue. Once an entry is removed from a block, it gets the entry from its left neighboring block, creating a right shift operation on the entire queue.

Each systolic array block consists of a holding register, which stores the entries in sorted order, as well as a temporary register, that holds passing entries enroute to the next block to the left. The passing entry is the lower-priority entry in a block during an enqueue operation. Multiplexors, a comparator, and decision logic also make up the rest of the block. A block diagram is shown in Figure 2.4. Queue capacity is increased by adding more blocks to the end of the queue without worrying about a central controller. Also, there is no bus loading problem as was the case with the shift register PQ. Increasing P requires extra storage and a wider comparator, as in the shift register priority queue. Unfortunately, the one main drawback is that the systolic array PQ requires twice more storage than the shift register architecture. Considering the simplicity of each block, the temporary register adds a considerable hardware cost to each block, compared to the shift register block. Also, the cost and delay of the comparator increases linearly with each extra bit in the priority field, which decreases the maximum operating clock frequency.

2.3 Modified Systolic Array Priority Queue

This section presents a new priority-queue architecture that combines the salient features of the systolic array and the shift register. The architecture has a tunable parameter which enables us to balance the trade-off between bus loading and hardware costs. We then extend this hybrid architecture to guarantee FIFO transmission of packets within the same priority level.

2.3.1 Hybrid Shift/Systolic

Of the four PQ architectures discussed in Section 2.2, the shift register architecture and the systolic array architecture are better than the other two in terms of supporting very large N and P. The FIFO architecture is limited to a small number of priority levels, while the binary tree comparator's complexity makes it difficult to scale with increasing N. On the other hand, the shift register and systolic array are more favorable because they have no centralized logic, and each block can be replicated as many times as necessary without any modifications. Also, a large number of priority levels can be easily supported by simply using more bits in the priority encoding. Unfortunately, the shift register's bus loading problem limits the maximum clock frequency, while the systolic array block's double storage requirement makes it considerably more hardware-intensive than the shift register.

The systolic array architecture scales well with N and its maximum operating clock frequency does not decrease as N increases. But, because 50% of all the registers are used as

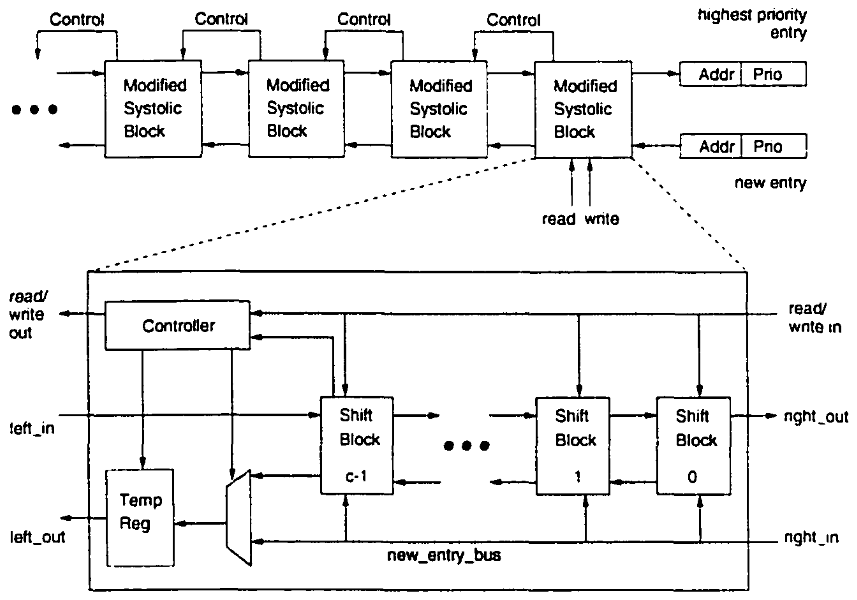


Figure 2.5: Modified systolic array priority queue

temporary registers, the systolic array uses much more hardware than the shift register. To reduce this overhead, we propose a modified systolic architecture where each block consists of a length **c** shift register. So instead of one temporary register for every holding register in each block, the ratio decreases by a factor of 1/c. Also, because scaling the modified systolic architecture for larger N does not involve changing c, the bus loading problem associated with the shift register stays constants as N grows.

Each modified systolic block holds **c** entries by replacing the single holding register with a length **c** shift register PQ, as shown in Figure 2.5. The interface of the modified systolic block is the same as that of the systolic block. Enqueue and dequeue requests are received from the right neighboring block and the results of those requests are sent to the right neighboring block. The right-most block receives requests and sends results to the link. During a new entry insertion into the modified systolic block, the new entry is placed in one of the blocks of the shift register PQ. If there is an overflow of the shift register PQ, either the new entry or the entry in the c^{th} shift block (whichever has lower priority) is placed into the temporary register and inserted into the left neighboring modified systolic block during the next cycle. Since the shift register PQ stores all the entries in sorted order with the highest-

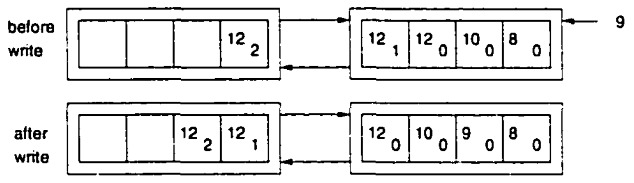


Figure 2.6: Data movement in the modified systolic array PQ showing a potential ordering problem

priority entry in the first block, the removal request is satisfied by moving all the entries one block to the right. The entry in the right-most block is sent to the neighboring right modified systolic block. During the next cycle, a removal request is made to the neighboring left modified block and the resulting entry is stored in the shift register PQ.

2.3.2 FIFO Ordering

Without any further modifications, the modified systolic array PQ will not maintain FIFO ordering among entries of equal priority, as illustrated in Figure 2.6. Here the number represents the priority and the subindex (not part of the entry) represents the arrival order among entries with the same priority. Insertion of a new entry with priority 9 pushes the 12_1 entry to the next modified systolic block and is placed behind the 12_2 entry. The problem here is that the second shift block cannot determine if the 12_1 entry corresponds to a new entry (which should go after 12_2) or an old entry (which should stay ahead of 12_2). This is solved by adding a one bit field (new/old) to the end (least significant bit) of the priority field and is included as part of the priority number when priority comparisons are done. The new/old bit is added as the entry enters the priority queue, and is stripped off when the entry leaves the queue. New entries that are inserted into the queue have this bit set. Likewise, all entries that are stored in a shift block have this bit set. The bit is cleared when an entry that was already in a shift block is pushed into the temporary register and sent to the neighboring left modified systolic block.

The modified systolic architecture improves on the systolic architecture by lowering the percentage of total registers used for temporary storage. This reduction in hardware is accomplished without losing any of the advantages of the systolic architecture - simple block architecture (easily scaled for increasing N by adding new blocks to the end of the existing queue), no performance loss as more blocks are added to the queue, and constant-time (cycles) enqueue and dequeue operations. Also, because the bus driving the shift register blocks is broken up into small length-c parts, the bus load within each modified systolic block is not affected by the additional modified systolic blocks. So, once a value for c is determined, only one modified systolic block must be designed and optimized for performance and area. This block is then replicated as many times as necessary without any modifications.

2.4 Multiple Output Link Priority Queue

To further reduce the hardware complexity of packet switches, we extend the architecture from Section 3 to service multiple output links. For simplicity, we first present a multiple-link priority queue based on the shift-register architecture before generalizing the technique to the hybrid systolic/shift design. We then discuss how the architecture can provide a constant-time dequeue operation, while still differentiating between packets destined for different output links. To avoid overlapping multiple operations in a single systolic array block, the design includes a small, constant number of "wait states."

2.4.1 Multiple Shift Register Priority Queue

Given that a switch has a separate priority queue for each of its M (>1) output links, the total queue capacity is MN entries. Since the shared buffer can only hold N packets, most of the blocks in the priority are unused at any given moment, as shown in Figure 2.7(a). An N-entry priority queue which services M (<N) output links can potentially save a maximum of 50% in hardware for M=2, and up to 75% for M=4. Here we present a multiple output link priority queue architecture, which has good scaling properties and constant-time enqueue and dequeue operations which are independent of M and N.

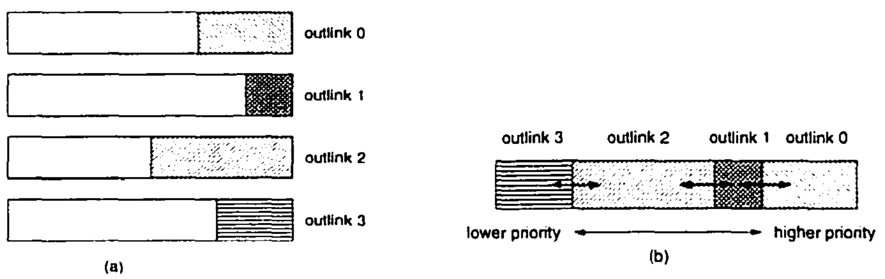


Figure 2.7: (a) Sorted entries in separate priority queue showing wasted resources; (b) Same entries in the multiple shift register priority queue

We first extend the shift register architecture to support multiple links. This requires modifications to the entries and shift register block. The packet's entry is augmented such that the priority field consists of the output link number, priority number, and new/old bit. The shift register stores entries such that those corresponding to higher output link numbers come after those corresponding to lower output link numbers, as shown in Figure 2.7(b).

The blocks in the shift register architecture require several modifications to support multiple output links. First, each block receives another control signal (outnum) which indicates the requested output link number. The value on outnum is latched along with the new entry during an enqueue operation, while it is used to determine which entry to output during a dequeue operation. Second, each block has a tristate buffer, which drives an output bus. This tristate buffer is needed because the highest-priority entry for a given output link can be in any of the blocks in the shift register. On a dequeue operation, a block will drive the output bus with the value in its holding register if the block decides it has the highest-priority entry for the requested output link. Figure 2.8 shows the block diagram of the multiple shift register queue with just the added control signals.

Within each multiple shift register block, no extra control logic is required for the enqueue operation. But during the dequeue operation each block needs to decide if it must drive the output bus. As seen from Figure 2.7(b), the highest-priority entry of any output link is always to the right of all other entries with the same output link number. Once the output bus has been read, all entries to the left of the one just read move one block to the right. The decision-making process for this dequeue operation is shown in Figure 2.9(a). A

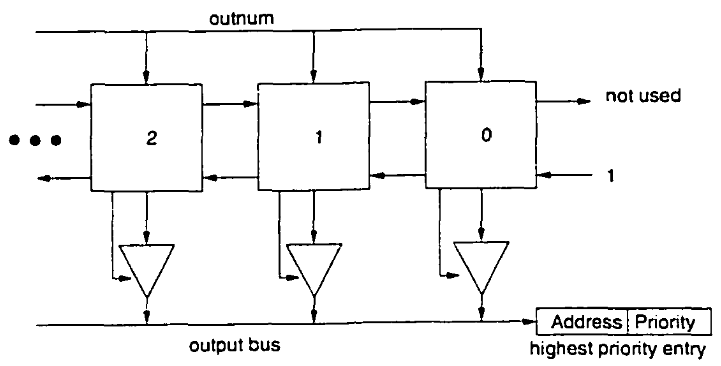


Figure 2.8: Multiple shift register priority queue

```
if read {
  if ((holding_reg_outnum == outnum) &&
     (rght_blk_holdng_reg_outnum < outnum)) {
    drive output bus;
    load left shift blk's entry on next cycle;
  }
  else if ((holding_reg_outnum >= outnum) &&
          (rght_blk_holdng_reg_outnum >= outnum)) {
    load left shift blk's entry on next cycle;
  }
  else {/*holding_reg_outnum < outnum*/
    do nothing;
  }
}
```

(a) Remove highest-priority entry

```
if read_low {
  if ((holding_reg_outnum == outnum) &&
     (left_blk_holdng_reg_outnum > outnum)) {
    drive output bus;
    load left shift blk's entry on next cycle;
  }
  else if (holding_reg_outnum > outnum) {
    load left shift blk's entry on next cycle;
  }
  else { /*holding_reg_outnum < outnum*/
    do nothing;
  }
}
```

(b) Remove lowest-priority entry

Figure 2.9: Pseudo code for read and read lowest-priority operations in multiple shift register block

similar operation can also remove the lowest-priority entry for an outgoing link, as shown in Figure 2.9(b). This operation is useful when extending the modified systolic architecture to support multiple outgoing links, as explained in the next subsection.

2.4.2 Multiple Systolic Array Priority Queue

Due to the bus loading problem in the shift register architecture, the PQ described in Section 2.4.1 does not scale well with respect to N. Besides the new entry bus, shown in Figure 2.3, the multiple shift register architecture also has the problem of each shift register block

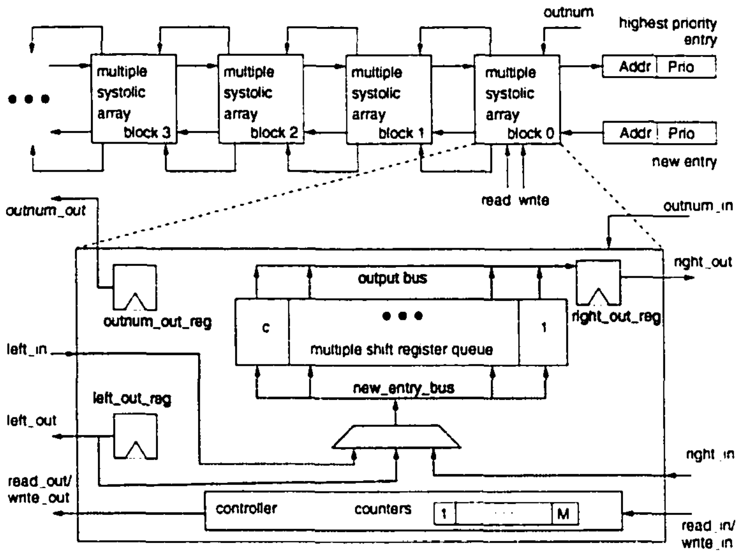


Figure 2.10: Multiple systolic array priority queue and block

driving the output bus, and the associated delay and hardware costs of having to drive a very large bus. Despite this problem, the multiple shift register can be used as a building block to support multiple outgoing links in the modified systolic architecture. By using the same ideas as in Section 2.3, the multiple systolic array architecture replaces the single holding register with the multiple shift register. By choosing a value for c which minimizes the total number of temporary registers without introducing significant bus loading problems, a single c-entry multiple shift register can be designed and used in the multiple systolic array architecture.

As seen in Figure 2.10, the external interface to the multiple systolic array block remains the same, with the addition of the outnum control signals. Besides the temporary register (left_out_reg), there is also another register (onum_out_reg) which indicates the link number of the entry in the temporary register. The right out register (right_out_reg) stores the output from the output bus, while a multiplexor chooses among three sources to drive the new entry bus. Also, instead of the read and write control signals directly feeding the shift register, the controller uses them to generate its own internal read and write control signals which are then fed to the shift register.

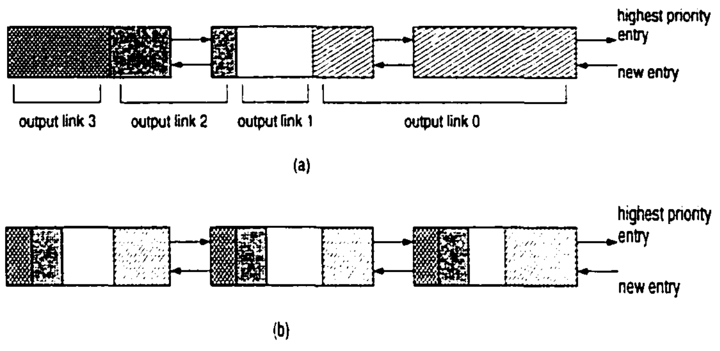


Figure 2.11: Storage of entries in the multiple systolic array queue (a) before and (b) after "atleast-1-entry-per-outputlink" property

2.4.3 Constant-Time Dequeue/Enqueue

Without further modifications to the architecture described in Section 2.4.2, a situation as shown in Figure 2.11(a) can occur. If there are more than c entries in the queue for any output link, a dequeue request can result in extra remove requests being sent from the one systolic block to the next systolic block. In the worst case, the requests can propagate to the last block, in which case the result will need to propagate all the way back up. In order to avoid this problem and have a constant-time dequeue operation, each systolic block uses counter to maintain a "atleast-1-entry-per-output-link" property, whenever possible. This assumes that $c \geq M$. A counter is used for each output link to keep track of the number of packets queued at that link. The counter is incremented (decremented) whenever an entry corresponding to the counter's output link is inserted (removed) from the systolic block.

After an enqueue operation, entries start to propagate through the array of systolic blocks in the left direction. Writing another entry into a systolic block that is full will result in either the right_in entry or the entry in the shift register queue's c block (which ever has lower priority) to be written into the left_out register. But before the entry in the left_out register is sent to the left systolic block, the controller makes sure that doing so does not violate the "atleast-1-entry-per-output-link" property. If it does, another entry is chosen to be sent to the left systolic block while the entry from the left_out register is reinserted into the shift register queue. Here the other entry that is chosen is the lowest-priority entry cor-

```
counter[outnum_in}++;
left_out=lower priority between right_in and
        shift-register-queue[c];
outnum_out=outnum of lower priority between
        right_in and shit-register-queue[c];
if (left_out is valid) do
  if (counter[outnum_dout]==1) do
    var outnum_temp = outnum such that
        counter[outnum_temp] > 1;
    right_out=lowest priority entry in shift
            register queue with outnum_temp;
    insert left_out into shift register queue;
    left_out=right_out;
    outnum_out=outnum of entry in right_out;
    invalidate right_out;
    coiunter{outnum_out}--;
    write left_out and outnum_out into left
        systolic block;
  enddo
  else do
    counter[out_out]--;
    write left_out and outnum_out into left
      systolic block;
  enddo
enddo
```

(a) write

```
if (counter[outnum_in]==0) do
  invalidate right_out;
enddo
else do
  counter[outnum_in]--;
  right_out=highest priority entry in shift
      register queue with outnum_in;
  outnum_out=outnum_in;
  request read from left systolic block;
  if (left_in is valid) do
    insert left_in into shift register queue;
    counter[outnum_in]++;
  enddo
enddo
```

(b) read

Figure 2.12: Pseudo code for write and read operations in systolic block

responding to an output link with more than one entry in the systolic block. The controller obtains this replacement entry by checking all the counters, and then issues a read_low command to the shift register queue. Also, following a dequeue operation, the zeroth systolic block requests an entry with the same output link number from the first systolic block which, after sending the result, requests an entry with the same output link number from the second systolic block, and so on. This is done to maintain the property for all systolic blocks. Details of these systolic block operations are shown in Figure 2.12. Note that the state transitions are not shown there.

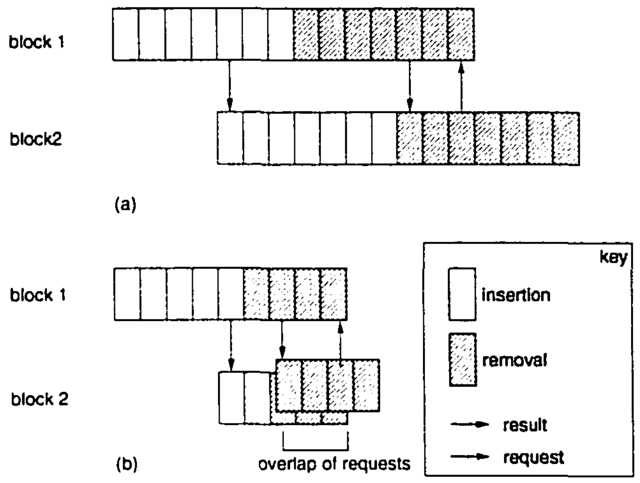


Figure 2.13: Time line of operation for multiple systolic block (a) with wait states (b) without wait states

2.4.4 Non-Overlapping Operations

Since the systolic block must finish one request before processing another request, wait states are needed to prevent overlapping of operations. The insertion operation takes 5 clock cycles, while the remove operation takes 4 clock cycles (these are independent of c and M). Without any wait states, a block can make an insertion request to its left neighbor on the 5th cycle while servicing an insertion request. Similarly, a remove request can be made to the left block on the 2nd cycle while servicing a remove request. Doing this will result in the overlapping of request service, as shown in Figure 2.13. This overlap is avoided by delaying the remove request till the 5th cycle, instead of the 2nd cycle. Since 2 cycles are need to get the result, a total of 7 cycles are needed for the remove request. Although the insert operation still takes just 5 cycles, consecutive requests can only be made every 7 cycles to the multiple systolic block.

Despite the added complexity of the state machine and extra hardware needed to support multiple output links, the multiple systolic array is still much cheaper to implement than individual priority queues for each output link. Also, the time (cycles) required to service the dequeue and enqueue operations is constant for any output link, and remains unchanged regardless of how large N becomes. Like the modified systolic architecture, each block is self-contained and no outside controller is required. As N increases, more blocks are added to the existing chain without modifications to the existing blocks. Also, since the priority number is encoded within each entry, a large number of priority levels can be supported without requiring a large amount of hardware. Thus, scaling does not involve modifying the architecture, implementation for large N is simplified since only one systolic block needs to be designed, and there is no loss in performance due to scaling.

2.5 Performance and Implementation

To compare the various priority queue architectures discussed thus far, each architecture was implemented using the Verilog hardware description language and the Epoch silicon compiler, an automatic layout generator. This provides a common framework which makes the cost and performance comparisons more meaningful. The implementation results showed that both new architectures had better scaling properties in terms of performance and hardware costs than the four existing architectures. Also, the multiple-link architecture was shown to scale well with M, provide good performance, and offered considerable hardware savings in comparison to using a priority queue per-link approach.

2.5.1 Evaluation Methodology

Costs were measured in terms of amount of silicon area and the number of transistors used by the design, while performance was measured by the maximum clock speed and throughput (number of enqueue/dequeue operations completed per second). Throughput can be easily calculated by using the maximum clock speed and number of clock cycles needed by each operation. Maximum clock speed was calculated by doing a critical path analysis of the design, and determining the delay through these critical paths using Epoch's timing analyzer. All designs were structurally specified using parts from Epoch's Verilog library, while state machines and control logic were described in behavioral Verilog. Each of the layouts was compiled by Epoch, which uses standard cells to generate a layout, using a 1.2 μm CMOS technology. Although custom layout would give better results, we are more interested in comparing the scaling effects than in raw numbers. In other words, we want to look at the relative costs and performance of the various architectures as N and P increase. Also, note that our implementations were limited to a maximum of 1024 for N. This was due to insufficient workstation memory for performing the various simulations.

One final note regarding the implementations concerns the use of registers for storage of priority queue tags. Although other storage devices could have been used in the designs, we chose to use registers because they allowed us to quickly implement, scale and compare the various designs. Since the main goal of this work was to study the relative scaling effects of the various designs, other storage alternatives were not studied. For example, in the shift and systolic architectures, the single holding register per block can be replaced with a SRAM module with the capacity to hold several priority queue tags. This will reduce hardware costs since the comparison and other logic is shared among several tags instead of one. But doing so increases the time required for dequeue and enqueue operations since these need to be serialized due to accesses to SRAM. So, choosing the size of the SRAM becomes a problem of sacrificing performance for smaller hardware costs. Although this work does not consider the trade-off between serial SRAMs and parallel registers, we have evaluated the associated cost and performance implications in the priority queue architectures in [59].

2.5.2 Existing Architectures

Figure 2.14 compares the four existing priority queue architectures in terms of VLSI hardware costs as a function of N, with P fixed at 16. Here we chose a small value of P for two reasons. It allowed for implementations with large N, and made the scaling effects associated with large N more pronounced. As expected, we see the systolic array architecture's hardware cost is much larger than that of the shift register due to the extra register used for temporary storage. Also, despite having similar transistor counts, the binary tree architecture occupies more area than the shift register architecture. This is mainly because of the routing required from the storage to the priority comparator tree, and routing within the comparator tree.

As expected, performance degrades with increasing N, as shown in Figure 2.15. Here we see the throughput is highest for the shift register architecture. But as N increases, the

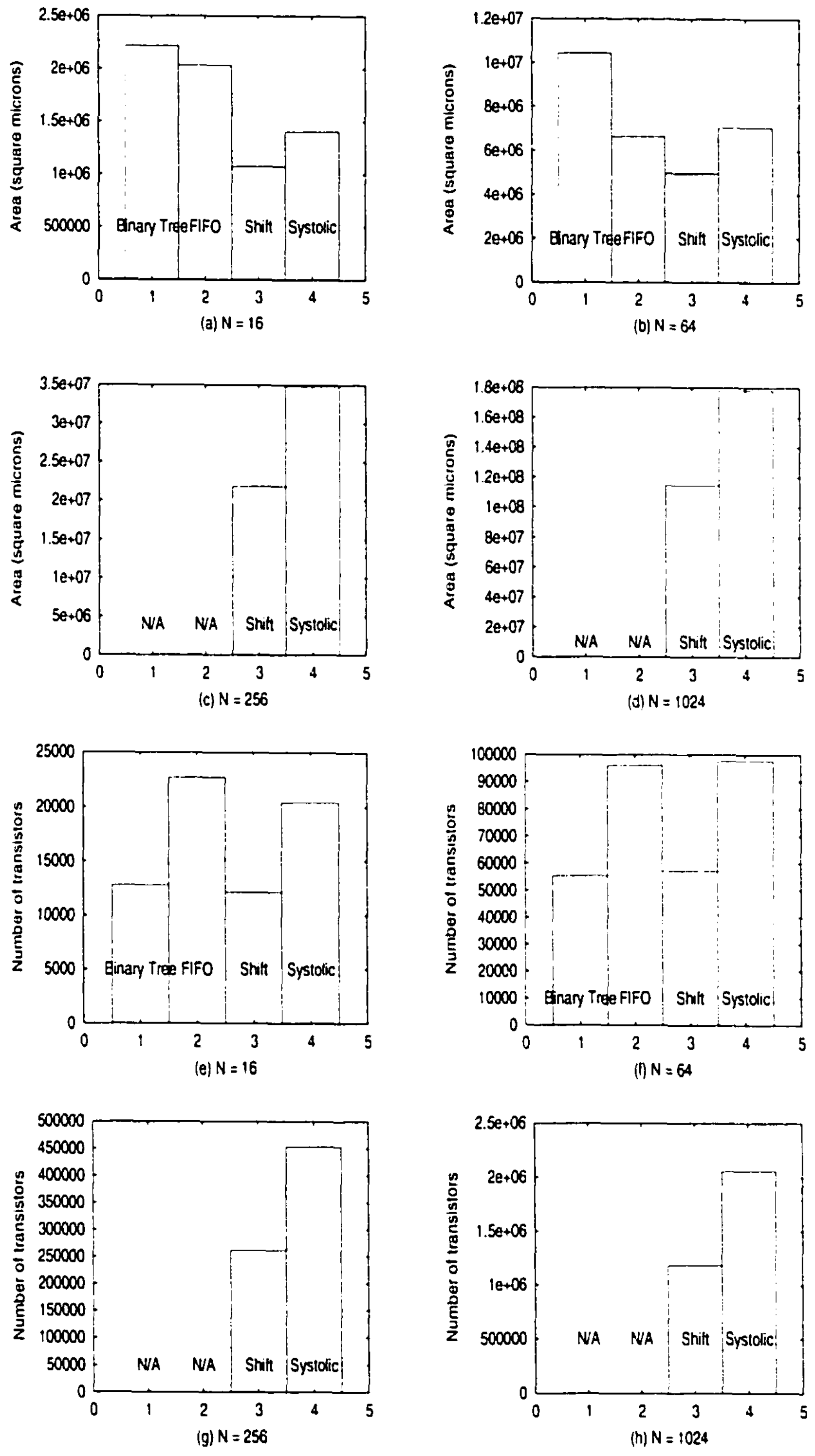


Figure 2.14: Implementation results for existing priority queue architectures (P=16)

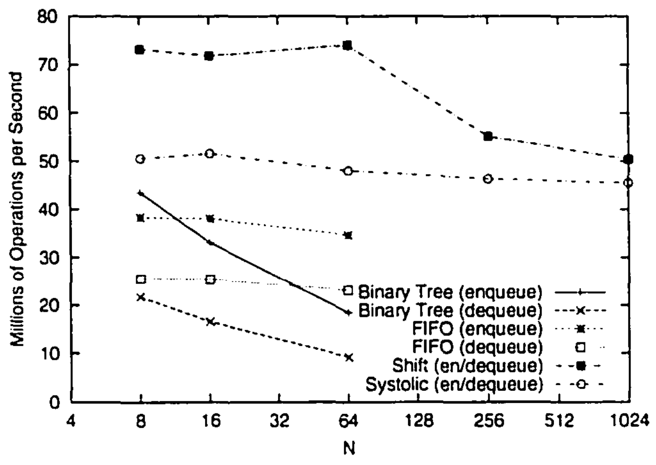


Figure 2.15: Scaling effects on performance as N increases (P=16)

performance degradation is much steeper for the shift and binary tree architectures than that of the systolic and FIFO architectures. This is due to the bus loading problem in the shift register and binary tree architecture, and the increase in depth of the comparator tree in the binary tree architecture. The gradual decrease in performance in the systolic and FIFO architectures can be attributed mainly to the extra bits in the registers and multiplexors, which add delay to the control signals which must drive these components. Although Figure 2.15 shows the shift register architecture with better throughput than the systolic array architecture, for larger values of N, we can predict the throughput of the systolic to be higher than the shift. Due to insufficient workstation memory we could not obtain data for larger values of N other than the ones shown in Figure 2.15. But by extrapolating the curves for the shift and systolic in Figure 2.15, we can see the two curves should cross at a point somewhere between N=1024 and N=2048. At this point, throughput of the systolic should be higher, while the performance of the shift architecture should continue to degrade at a much faster rate than that of the systolic due to the dominating effect of the bus loading problem. For much larger values of N, this bus problem should make the shift register architecture an ineffective solution due to the associated hardware costs and performance loss.

Each bit added to the priority field adds delay to the priority comparator, which in turn slows down the operation of the priority queue for the shift register, systolic array, and

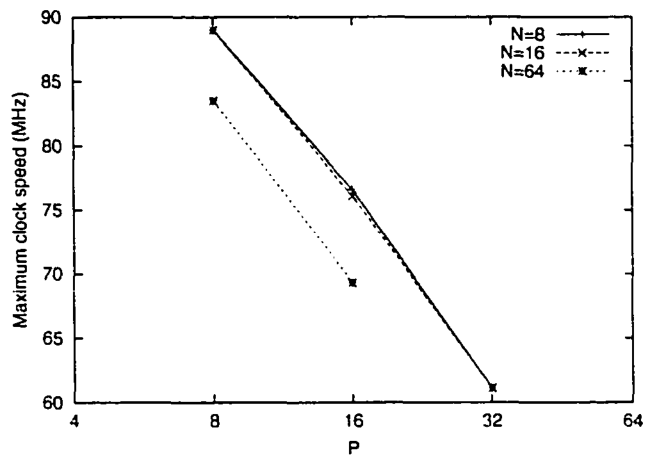


Figure 2.16: Scaling effects on FIFO architecture as P increases.

binary tree architectures. Since a large number of priority levels can be supported with relatively few bits, and because the delay associated with the extra bit is small compared to the total delay, scaling for large P is feasible and the resulting implementations can be effective. With a non-pipelined binary tree though, the delay is multiplied by the depth of the tree. In the FIFO case, the bottleneck is in the priority encoder, which must scan each FIFO to select the next highest priority entry. This can be seen in Figure 2.16. Note also that the depth of the physical FIFO (due to increasing N) does not affect performance, but adds to the FIFO fall-through time. So, it is possible that an entry might not be available immediately after it is inserted into the queue. The logical FIFO architecture avoids this problem by using linked lists instead.

2.5.3 Modified Systolic Array Architecture

The motivation for the modified systolic array architecture was to take advantage of the shift register and systolic array architecture's features and, at the same time, reduce the negative side-effects due to scaling with respect to N. The shift register architecture suffered from the bus loading problem, while the systolic array architecture used a significant amount of extra hardware for the extra register. The solution that was proposed was to use a separate shift register queue inside each systolic array block. Each shift register queue stores c entries, where c is determined by hardware and performance requirements. When

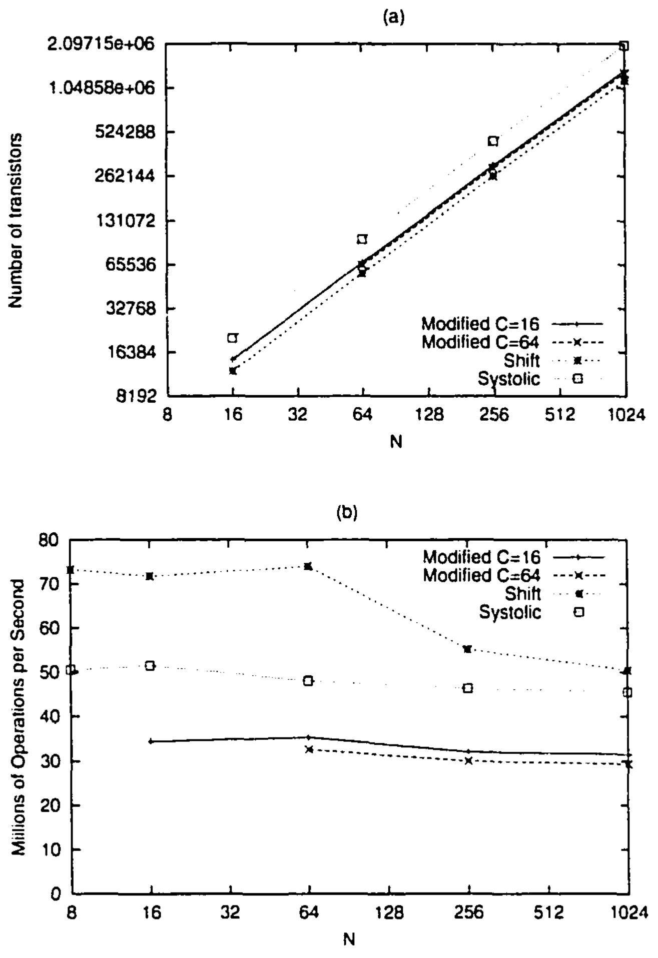


Figure 2.17: Modified systolic architecture results compared to that of the shift and systolic (P=16)

c=1, this is the same as the original systolic architecture, whereas if c=N, then we get the original shift register queue. So for small c, hardware costs and performance are close to those of the systolic architecture, and as c increases the hardware costs steadily approach those of the shift register architecture. Also, as c increases, the performance of the modified decreases due to extra bus loading, as is the case with the shift architecture. This point is shown in Figure 2.17. Here P=16, and two values of c are used. Initially for small values of N, the shift architecture has the best performance, mainly because of negligible bus loading and also because the operations in the shift architecture require one cycle, as opposed to two in the systolic and modified. But as N increases, the rate at which performance decreases is much sharper in the shift register case due to the bus loading problem. With the systolic and modified systolic, the performance curve is relatively horizontal. For larger N, performance for the modified systolic should be higher than that of the shift register due to this very gradual decrease in performance in the modified systolic architecture. Considering the amount of hardware used in the modified systolic is only slightly more than that of the shift, and considering the aggressive buffering strategies required in the shift architecture to get these performance numbers, the modified systolic is a much more effective solution despite the performance difference. So once a value for c is determined based on the performance requirements, the design can be scaled to very large N, without worrying about severe performance degradation from bus loading, or buffering strategies.

2.5.4 Multiple Systolic Array Architecture

Despite added hardware costs (due to extra registers, added complexity of control logic, tristate buffers, and counters), we see that there is still a substantial amount of hardware saved by using the multiple queue. Based on implementations with a 16-entry multiple systolic array block, we observed the following. For M=4, the multiple architecture occupied 32% less area and used 55% less transistors versus the shift, and 46% less area and 72% less transistors versus the systolic. For M=8, the multiple architecture occupied 67% less area and used 75% less transistors versus the shift, and 73% less area and 85% less transistors versus the systolic. Here we multiplied the costs for a single shift or systolic queue by M to account for one queue per output link. Some results are shown in Figure 2.18. We also observed that adding support for more output links in the multiple systolic block increased the costs only slightly. This is because most of the multiple link support already exists, and all that is needed are extra counters and minor additions in the controller. This can be seen by the extra line inside the bars for the multiple systolic architecture in the M=8 graphs. The top line indicates the value for M=8, while the lower line shows the value for M=4. As seen, the difference in the two lines is very small indicating a small increase in cost for extra output link support.

For c=16, N=64, and P=256, the maximum clock speeds for the multiple systolic architecture are 40 MHz (M=4) and 38 MHz (M=8). This drop in speed is due to the extra bits in the priority field used to encode the output link number. Considering each enqueue and

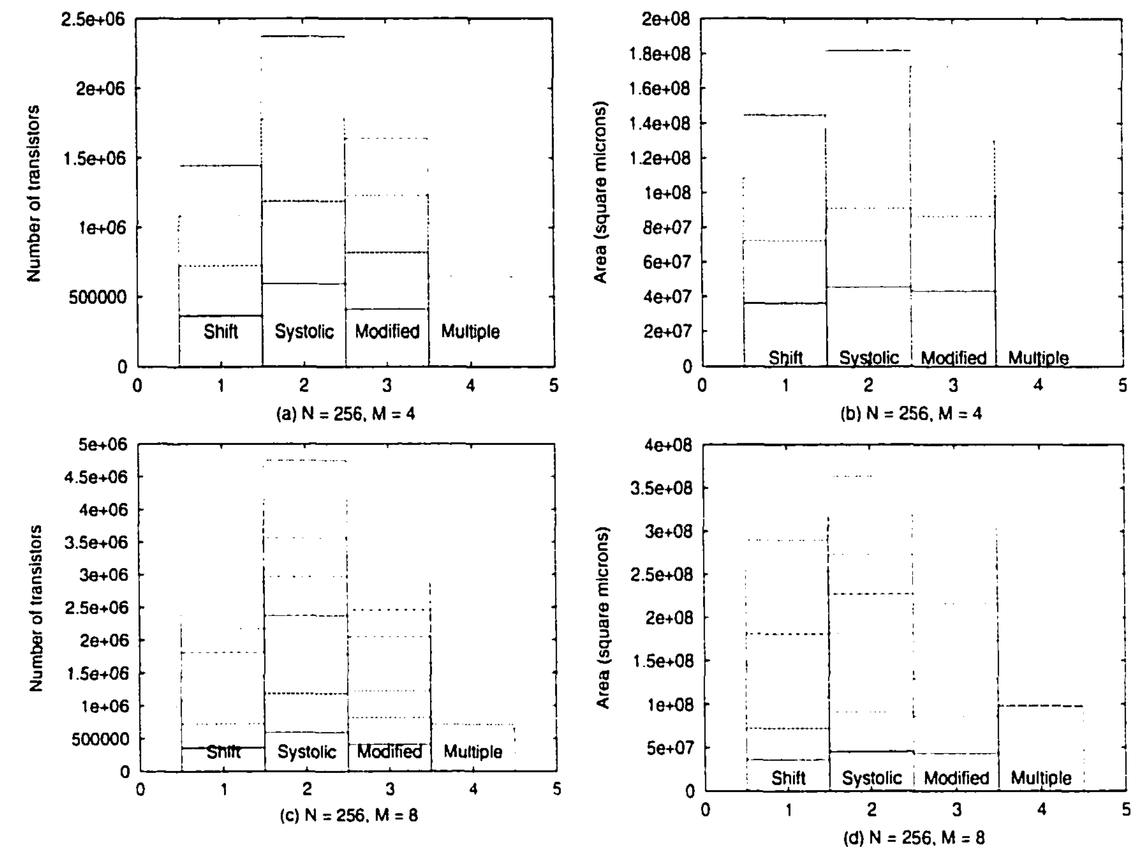


Figure 2.18: Implementation comparison with multiple systolic architecture (P=256)

dequeue operation requires 7 cycles, this translates into 5.71 mops (millions of operations per second) for M=4, and 5.43 mops for M=8. If we consider each switch as having M inputs and M outputs, with all input and output links getting round-robin access to the queue, the queue can support link speeds up to 303 Mbps for M=4, and 144 Mbps for M=8 (assuming 53 byte packets). At current ATM standards of 155 Mbps, a multiple systolic priority queue can be designed and implemented to support such switches. For switches with a larger number of links, by grouping 4 to 8 outgoing links together, hardware costs can still be significantly reduced while being able to support very high-speed links.

2.6 Summary

In this chapter we proposed and evaluated two new hardware priority queue architectures for link scheduling in high-speed switches. Based on Verilog and Epoch designs and simulations, we showed that the four existing architectures were limited by scalability (with respect to either N or P or both). For small N and P, all four existing architectures had comparable hardware costs and performance. But as they were scaled to support large N and P, each architecture's limitations became more pronounced. Of the four architectures, the shift register architecture and the systolic array architecture had better scalability. By combining the two architectures, the modified systolic architecture reduced the negative effects of scaling suffered by the two architectures. In particular, hardware costs were significantly reduced by decreasing the number of total temporary storage registers; performance loss due to the bus loading problem in the shift register could be controlled and isolated from N by using several length-c shift register queues. Here c was chosen by considering hardware and performance requirements. The multiple systolic architecture added multiple link support to the modified systolic architecture, without sacrificing scalability. Although extra cycles were added to the dequeue and enqueue operations, both these operations could be done in constant time (cycles), regardless of N or M, the number of output links supported by the architecture. We have also observed that scaling with respect to M was possible with very little additional hardware. Verilog and Epoch simulations have confirmed the salient features of the new architectures.

We showed that the two new hardware priority queue architectures scale well to increasing N and P. Both offer good performance and are easy to implement, and hence can be used in guaranteeing QoS requirements in high-speed networks. Such effective priority queue implementations allow switches to use more aggressive link-scheduling algorithms that can admit more connections with diverse traffic patterns and QoS requirements.

CHAPTER 3

INTEGRATED TRAFFIC SHAPING AND LINK SCHEDULING ARCHITECTURES

3.1 Introduction

This chapter examines the movement of tags (which correspond to a packet in the shared buffer) within a shaper-scheduler. We examine several traffic shaping and link scheduling algorithms to determine the basic mechanisms required in an implementation. Building on the priority queue architectures proposed in the previous chapter, we present two new shaper and scheduler architectures which were designed for high-performance and flexibility with good scaling properties with respect to N, the number of flows a shaper-scheduler must keep track of for a given output link. The first architecture presents a simple solution which allows for a constant-time operation to find and move an eligible packet from the shaper to the scheduler queue. This is accomplished by extending the multiple-link priority queue architecture presented in Section 2.4. The second architecture eliminates the shaper to scheduler problem by using a single-stage architecture instead of traditional two-stage shaper and scheduler architectures (the concept of one-stage and two-stage architectures is discussed in Section 3.2). This is accomplished by using two multiple-link priority queues in parallel. The resulting architecture can provide the same functionality as traditional two-stage architectures while maintaining constant-time operations.

Before describing our new architectures, we first provide some background on traffic shaping and link scheduling in Section 3.2. We then describe some shaper/scheduler architectures from the current literature in Section 3.3. A brief description of each architecture and operation is given, followed by a discussion on limitations to their flexibility and scalability. The two new architectures are then presented in Section 3.4 and Section 3.5. Details of the architectures and operation are explained here, along with discussion of implementation issues and changes to the architecture to deal with implementation related scaling effects. Section 3.6 concludes with a summary of the chapter.

3.2 Background

There are numerous traffic shaping and link scheduling algorithms in the literature, each characterized by different QoS guarantee properties. Despite this difference, all these algorithms share a common framework upon which they can be mapped. Traffic shapers hold back newly-queued packets from being serviced until the packet's flow conforms to its traffic envelope. This implies marking each packet with a conformance (eligibility or start) time, and having the shaper move packets out of the shaper and into the scheduler queue only when the system time reaches the start time. Shapers differ in how they compute the start time. On the other hand, when the link becomes available, the link scheduler chooses the next packet to transmit among all eligible packets queued in the scheduler queue. This implies that packets are assigned a priority (also called *deadline* or *finish time*), with the scheduler choosing the packet with the highest priority. Scheduling algorithms simply differ in how they compute the priority. We illustrate these points by describing several well-known algorithms.

3.2.1 Leaky-bucket Shaper

This algorithm [19] [58] [59] [75] [78] is conceptually very simple and is the basis for most shapers in the literature. The shaper generates tokens for a flow i at a rate of ρ_i, where σ_i is the maximum number of tokens that can be accumulated in the bucket. A newly-arrived packet is eligible for transmission only if there are enough tokens in the bucket. Otherwise, it must wait in the shaper until enough tokens have accumulated. When a packet is eligible and moves into the scheduler, it grabs the necessary number of tokens from the bucket. In the case of fixed-sized packets (i.e., ATM cells), each token corresponds to a single packet. Assuming the k^{th} packet from flow i arrives at time $A(p_i^k)$, and requires L_i^k tokens, the packet's start time $S(p_i^k)$ (earliest time the packet is eligible) can be computed very easily.

Since the shaper will serve packets from the same flow in FCFS order, the shaper need only keep one entry for each flow with one or more packets queued in the shaper. The packets can be ordered as a simple list. When the first, or head-of-the-line (HOL) packet becomes eligible, the start time for the flow's next queued packet is calculated by setting its arrival time as the start time of the previous packet. If there are packets queued in the shaper for flow i when a new packet arrives, the new packet is simply appended to the end of the list without the need to calculate its start time.

3.2.2 Link Scheduling

In a static priority algorithm each flow has a predetermined priority number associated with it. All packets belonging to the flow are marked with the same priority. The special case where there is only one priority level is the FCFS algorithm, where no packets have priority over others.

Under EDD [2] [30] [39] [59] [75] [81] [89], each packet is assigned a due date (deadline), with the scheduler transmitting smallest deadline first. With these schemes each flow i provides the minimum packet interarrival time I_i and a local delay bound d for each node the packet passes in the network.

Packet-by-packet generalized processor sharing (PGPS) [52], also known as Weighted FQ (WFQ), Frame-based FQ (FFQ) [69] [80], Starting-potential based FQ (SPFQ) [68] [80], Start-time FQ (SFQ) [32], Worst-case Fair WFQ (WF2Q) [6] and WF2Q+[7], and Self-clocked FQ (SCFQ) [31] are several examples of packetized versions of fair queueing (FQ) algorithms [19] [22] [52] [86] [89]. Despite the large number of variations, the basic foundation for all these algorithms is the same. The function $V(t)$ returns the system time (or system potential) at time t. For each packet that enters the scheduler, it is assigned a start time $S(p_i^k)$ and a finish time $F(p_i^k)$. Packets are then transmitted in increasing order of start time or finish time, depending on the algorithm. The difference among the various PFQ algorithms lies in how they compute the system, start, and finish times. Typically, each flow is specified by only by its allocated rate ρ_i.

Other link scheduling algorithms include weighted round-robin [41], hiearchical round-robin [38], Stop and Go [76] (a framing strategy), Virtual Clock [90], and many others.

3.2.3 Framework

The common denominator among all these algorithms is the notion of stamping each packet with a number, with service order based on that number. Shapers queue packets and service them based on their start times, while the scheduler queues packets based on their priorities. Packets in the shaper queue can only be considered for transmission when they are eligible (i.e., the shaper moves the packet to the scheduler queue). Several researchers have proposed various shaper-scheduler implementations [14] [48] [70], whose results we incorporate into our work. The basic framework of these implementations includes two sorted queues (shaper and scheduler), a mechanism for determining and moving eligible packets into the scheduler queue, and a control mechanism which computes the start and finish times of each packet. After each packet transmission, the shaper needs to compute the start times of any new HOL packets that have arrived and insert them into the shaper queue. Next, the shaper must move any eligible HOL packets into the scheduler queue. If these HOL packets are not the last in the flow's linked list, then the new HOL packets must be processed by the shaper and inserted into the shaper queue. The scheduler then needs to determine and transmit the next packet.

3.3 Integrated Shaper and Scheduler Architectures

Figure 3.1 shows a generic view of the shaping and scheduling mechanism. Since packets from the same flow are serviced in FCFS order, the shaper performs per-flow shaping and need only maintain one entry per flow in the shaper queue. Packets in a flow are maintained as a linked list, with new packets appended to the end of the list. Each output link processor has a separate linked list manager, which maintains the linked list for each flow. This linked list manager is typically incorporated into the shaper-scheduler design. Only the eligible time of the HOL packet in each flow is used by the shaper. Eligible times for the remaining packets in the flow can be easily computed from the eligible time of the previous packet.

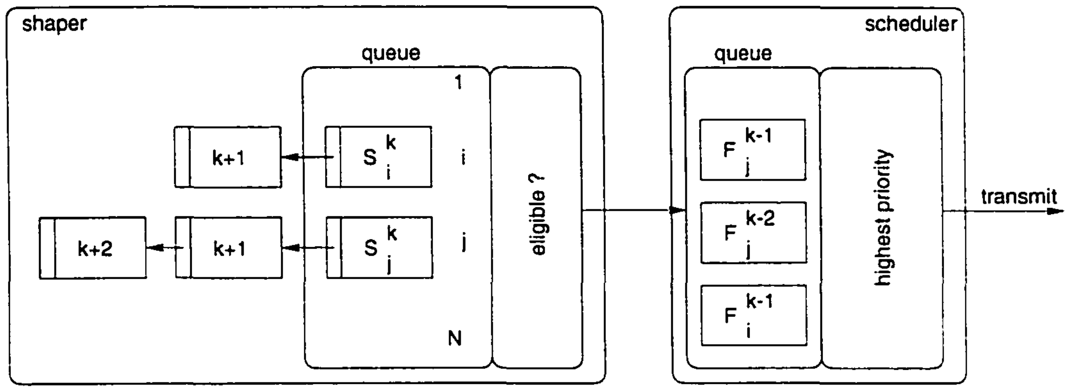


Figure 3.1: Logical view of a two-stage shaper and scheduler architecture, where N=max number of flows

This assumption holds because $S(p_i^k) \geq S(p_i^{k-1})$ and $F(p_i^k) \geq F(p_i^{k-1}$. Based on the system clock, the shaper calculates the finish times for all eligible HOL packets and moves them into the scheduler. Unlike the shaper, the scheduler performs per-packet queueing, and services packets in order of their priorities. After moving the HOL packet, if there are more packets in the flow the shaper computes the start time for the next packet in the list and places it into the shaper queue.

This type of two-stage architecture [87] [88] (first-stage shaper and second-stage scheduler) lends itself well to implementing a combination of shaping and scheduling algorithms. This is because all shaping and scheduling algorithms described in Section 3.2 can be mapped into an implementation which produces an eligible time (shaping), a finish time (scheduling), or both (S-RPS). A network architect can mix and match algorithms to meet desired QoS goals. However, this type of flexibility requires that the architecture provide the following basic operations needed by all the algorithms. First, the shaper requires an efficient queueing mechanism to store start times for all HOL packets and quickly find all eligible packets given the system time. Similarly, the scheduler also requires a queueing mechanism to store finish times and find the packet with the highest priority or the earliest finish time. Finally, an efficient mechanism is required to transfer all eligible packets from the shaper to the scheduler. In other words, an efficient integrated shaper and scheduler

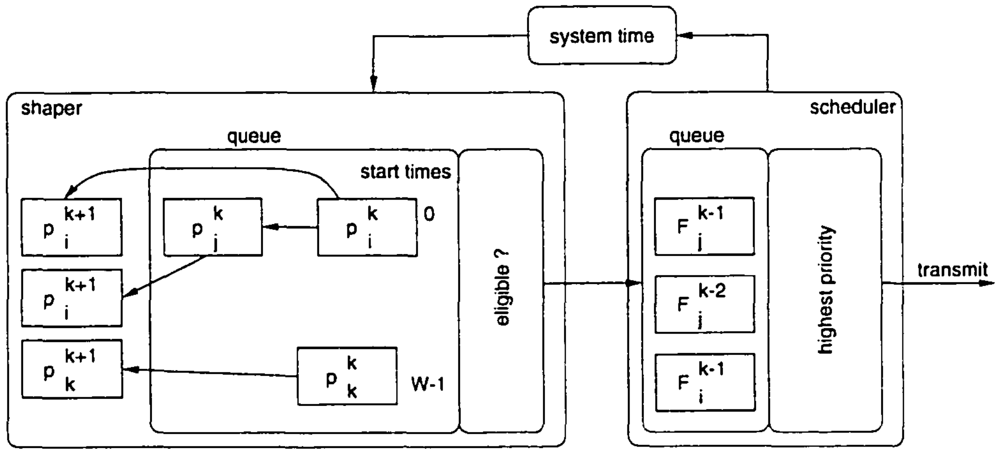


Figure 3.2: Logical view of shaper-scheduler with HOL packets arranged by their starting times.

architecture must provide an efficient solution to the sorting problem, and an organization of the sorters which can solve the shaper-to-scheduler transfer problem.

Since the actual packets are stored in a shared buffer, the shaper-scheduler only manipulates tags. Each queued packet has a corresponding tag in the shaper-scheduler. Throughout the rest of this thesis, the notion of queueing, sorting, and moving of packets by the shaper-scheduler is used to refer to the manipulation of the tags corresponding to the packets.

The rest of this section describes and evaluates several two-stage architectures from the current literature. Section 3.3.1 describes an architecture [14] which uses a search-based priority queue engine (RSE) to solve the sorting problem, and uses several of the RSEs to implement an integrated shaper and scheduler. A calendar queue [18] is used in [67] [70] [80] to implement the shaper queue. This is described in Section 3.3.2. To reduce implementation costs an approximation scheme using groups of FIFOs [58] is described in Section 3.3.3.

3.3.1 Search-based Sorting

A logical view of the shaper-scheduler architecture proposed in [14] is shown in Figure 3.2. HOL packets are maintained according to their start times in the shaper queue. Since the shaper can transfer only one packet at a time to the scheduler, HOL packets with the same start time must also be sorted in order of their finish times. This requires that the shaper sort packets based on their start times and finish times. Since the number of packets with the same start time is unbounded, the time it takes for the shaper to transfer all the packets with the same start time (*S*) could be potentially much larger than the packet transmission time. During this time, the system clock will continue to count up and other packets could also become eligible. At this point packets with start times larger than *S* will continue to wait even though some of the packets might have finish times which are smaller than those packets with start time *S*. As described in [14] [70], the shaper only needs to transfer two packets to the scheduler (among packets with start time equal to the current system time, the packet with the smaller finish time, and among packets with start time equal to the start time of the packet just transmitted, the packet with the smallest finish time) during each time slot. The reason for this is because none of the other eligible packets in the shaper queue will transmit during the next time slot since their finish times are greater than or equal to that of the packet with the smallest finish time in the scheduler queue. A *time slot* is defined as the time required to transmit a fixed size segment. A packet can consist of one segment (i.e., ATM cell) or multiple segments.

Instead of a sorted priority queue (PQ), this architecture uses a RAM-based search engine (RSE) as its basic PQ building block. Given an array $[0, \ldots, W-1]$ whose values are either 0 or 1, the RSE returns the smallest index with value 1. In other words, the index represents the start/finish time, and a 1 indicates that there are packets with that start/finish time. In the case of the shaper queue, a 2D RSE is used, with each element of the 2D array indexed by [start time, finish time]. A value 1 indicates that there are HOL packets with those values. Given a start time, the 2D RSE returns an index number which is a concatenation of the start time and finish time corresponding to the smallest finish time with value 1. Assuming that the maximum value for the start time is *W* and the *M* for the finish time, the shaper requires an array of length *WM*. The scheduler queue requires a 1D RSE with an array of length *M*.

The main disadvantage of this architecture is that scalability is dominated by *WM*. If we assume 15 bit values for both *W* and *M* and a maximum of N flows, the amount of memory required for the shaper array is $2^{15} \times 2^{15} \times \log_2 N = (134 \times \log_2 N)\text{MBytes}$. To support a large range of rates, the values for W and M could require more than 15 bits, making this an expensive implementation. Also, the RSE uses a tree-based algorithm whose implementation and time per search is dependent on the actual values of W and M, which makes it difficult to scale with respect to these parameters. The time for each RSE operation is $O(\log|RSE|)$, where $(\log|RSE|)$ is the depth of the tree used in the RSE.

This architecture assumes that the system clock is incremented after every segment transmission. When there are no eligible packets to transmit, the system clock is set to the smallest start time among all queued HOL packets in the shaper. Because the shaper queue simply returns the packet with the smallest finish time given a start time value, a separate RSE is used to track start times for all queued HOL packets, further adding to implementation cost.

Since each flow is maintained as a linked list of packets, implementation costs increase linearly with N. The rest of the shaper-scheduler and operations are not affected by the size of N. Also, because of its generic two-stage architecture, most of the algorithms described in Section 3.2 can be accommodated.

3.3.2 Calendar Queue Sorting

The architecture presented in [67] [70] [80] also implements the logical shaper-scheduler shown in Figure 3.2. The main difference is that a calendar queue [18] is used in the shaper to maintain start times of HOL packets. The shaper queue is implemented as an array $[0, \ldots, W-1]$, where W is the maximum value for the start time. Each element of the array points to a list of HOL packets with the corresponding start time. Since packets in each list can have different finish times, each list must sort packets in increasing finish time order. This sorting is necessary because of the two packet transfer per time slot rule described in the previous section.

The main disadvantage of this architecture is the need to sort each list each time a new packet is added. An $O(\log N)$ implementation is proposed which re-arranges the list in increasing finish time order. For high-speed networks this approach might not be fast enough to keep up with packet arrivals. An alternate approach could use a separate hard-

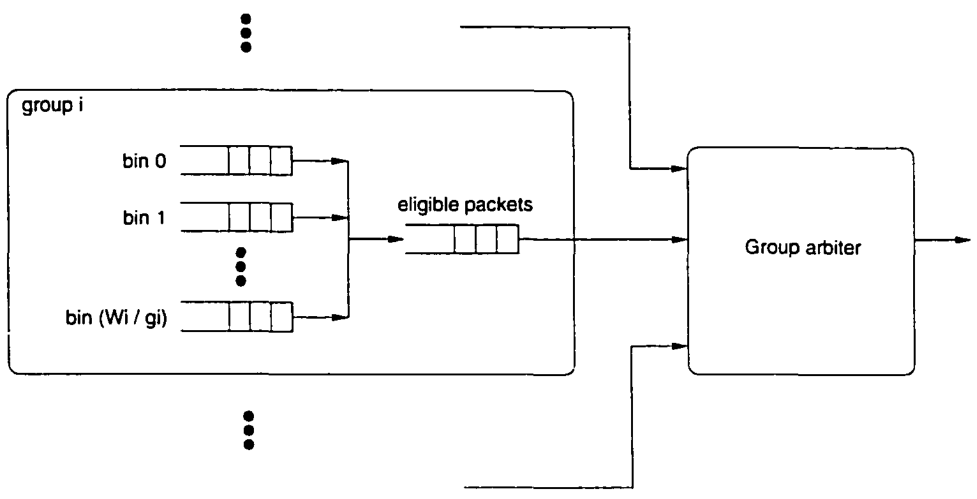


Figure 3.3: A hierarchical shaper-scheduler architecture with approximate sorting units.

ware PQ with constant time operations [48] for each list. This, however, requires *W* PQs of capacity N each (to account for the worst case when all flows have the same start time), resulting in very high implementation costs.

Because of the calendar queue implementation, a pointer scanning the array requires $O(W)$ time to find the smallest start time among all queued HOL packets in the shaper. Time required for this search can be reduced by using another PQ of start times, similar to what is done in [14]. This, however, adds to the implementation cost.

Scalability with respect to N and flexibility properties are the same as those of the architecture described in Section 3.3.1. The main difference is in the time required for each operation ($O(\log N)$ vs. $O(\log RSE)$) and the implementation cost for the shaper ($O(W)$ vs. $O(WM)$).

3.3.3 Approximate Sorting

[17] [58] [66] realize the high cost of sorting and propose an approximation scheme. Instead of maintaining a list of HOL packets sorted by their starting times, HOL packets are grouped together based on their allocated service rates ρ. As shown in Figure 3.3, [58] proposes a hierarchical arrangement of groups, where each group services a range of rates. Within each group a sorting unit places the packet into a sorting bin, where each bin holds

packets with start times which fall within a certain range. If W_i is the maximum start time for group *i* and g_i is the bin granularity in group *i*, then each group only needs W_i/g_i bins. Within each bin packets are serviced in FIFO order. Every g_i units of time, all the packets in a bin become eligible for transmission. The group arbiter then schedules transmission among eligible packets from all groups by using a Weighted RR scheme or a variation of the SCFQ described in [58]. The architectures in [17][66] place further restrictions by supporting only a finite number of rates.

Since each FIFO bin can be implemented as a simple linked list, the cost of each group is $O(W_i/g_i)$. However, this savings in implementation cost comes at a sacrifice of flexibility. Because these approximation schemes affect the performance guarantees of the original algorithm, careful analysis is needed to make sure that for a given service discipline the deviation is not too great due to the approximation. This severely limits the range of service disciplines supported by this architecture. For example, a strict EDD scheme would not be possible under this architecture without a very large number of bins, and a very fast search mechanism to find the non-empty bin with the smallest deadline.

3.4 Sorted Constant-time Two-Stage Architecture

This section presents a new two-stage shaper-scheduler architecture which provides constant time operations for both the shaper and scheduler which are not affected by *N*, *W*, or *M*, and has implementation costs which scale linearly to *N*. The constant time operation is accomplished by extending the PQ concepts shown in [48] to allow for the removal of the packet with the smallest finish time for a given start time. We show how this is used in the shaper-to-scheduler transfer. We also discuss implementation issues related to this architecture and present a solution.

3.4.1 Dual-Key Shift PQ

Of the shaper-scheduler architectures discussed in Section 3.2, the architectures with exact sorting on both the start time and finish time provide the most flexibility. However, their effectiveness depends on how efficiently they can find and transfer eligible packets from

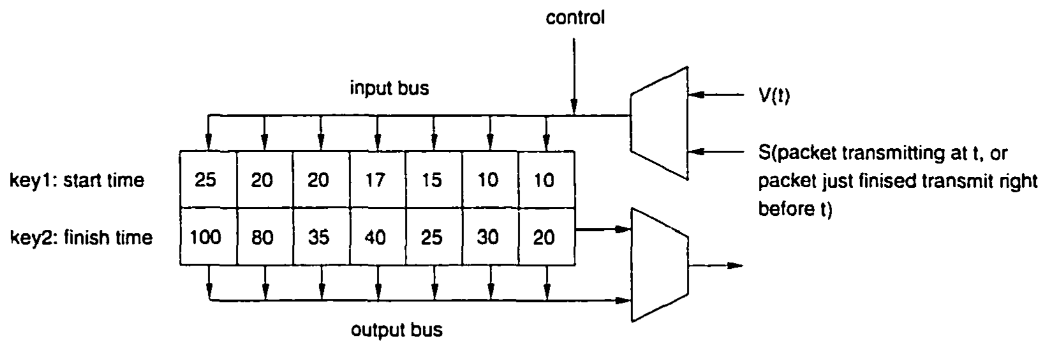


Figure 3.4: Logical block diagram of dual-key shift PQ.

the shaper queue to the scheduler. Assuming variable size packets, at each time slot at most two packets need to be transferred. A smaller time slot offers a wider range of rates to flows, but reduces the time allowed for the transfer. The $O(\log)$ solutions in Section 3.2 will not scale to increase in link speed and number of flows. A larger time slot can potentially cause the link to idle, since packets that do not arrive on time slot boundaries will have to wait until the next time slot even though they might be able to transmit otherwise.

We present a simple solution to the transfer problem by extending the multiple shift PQ presented in Section 2.4 so that it can be used as the shaper PQ. The main idea of the shift PQ [15] [48] [74] is as follows. The PQ consists of an array of blocks, each block holding one tag. There are N blocks, since only HOL packets are stored in the shaper PQ. At any given time the PQ maintains the tags in order of the priorities (i.e., block 0 stores the highest, block 1 the next highest, and so on). Assuming PQ blocks are numbered right to left (0 being the rightmost block), the net effect of writing a new tag into the PQ is to force all tags with lower priority than the new one to shift one block to the left, while the new tag places itself to the left of the tags with higher and equal priority. A readout of the highest priority tag from block 0 results in a shift of all tags one block to the right. By sorting on two keys instead of one, the multiple shift PQ is able to support more than one output link. A tri-state buffer is added to each block to enable readout of the highest priority tag (based on the *key2*) for a given output link (*key1*). In our shaper architecture, *key1* refers to the start time, while *key2* becomes the finish time. A logical block diagram is shown in Figure 3.4. As shown, tags with the same start time are ordered based on their finish times. Because each block is able to make the correct decision based solely on local information, the time

```
if read {
   valid(i) = 0;
   if (key1(i) == key1 on input_bus) and
      (key1(i-1) < key1 on input_bus) {
      valid(i) = 1;
      drive output_bus = key2(i);
      key1(i) = key1(i+1);/*this is done on rising edge of next*/
     key2(i) = key2(i+1);/*cycle, so read only need 1 cycle*/
                 /*this does a shift right starting at block i*/
    }
    else if (key1(i) >= key1 on input_bus) and
       (key1(i-1) >= key1 on input_bus) {
       key1(i) = key1(i+1);/*this does a shift right*/
       key2(i) = key2(i+1);/*starting at block j<i*/
    }
    else {
      do nothing; /*desired block j > i*/
    }
}
```

Figure 3.5: Pseudo code for read operation in block *i* of the dual-key shift PQ.

required for each write or read operation is 1 clock cycle. Figures 3.5 and 3.6 shows the control logic for each block in the dual-key shift PQ. Not shown is the shift operation, which moves the entire PQ one block to the right.

Given the start time on the input bus, only one block will drive the output bus with the result, if a tag exists with that start time. Detecting whether or not there is valid data on the output bus requires that each block also output a valid bit, with all valid bits logically ORed. A "1" indicates valid data on the output bus, and "0" otherwise.

3.4.2 Minimum Time Slot Value

The system time is updated at every time slot, at which time HOL packets that arrived during the previous time slot need to be inserted into the shaper queue and at most two packets need to be transferred from the shaper to the scheduler. In the worst-case scenario the number of HOL packets that need to be inserted will be *RX*, the number of input links of the node. Typically *RX* ranges from 4 to 128 for today's high-end commercial routers

```
if write {
   if (key1(i) > key1 input_bus) {
      key1(i) = key1(i-1);/*this is done on rising edge of
next*/
     key2(i) = key2(i-1);/*cycle, so read only need 1 cycle*/
              /*this does a left right starting at block j<i*/
   }
   else if (key1(i) == key1 input_bus) and
      (key2(i) > key2 input_bus) {
      key1(i) = key1(i-1);
       key2(i) = key2(i-1);
   }
   else if (key1(i) == key1 input_bus) and
      (key2(i) < key2 input_bus) {
      key1(i) = key1 input_bus;
      key2(i) = key2 input_bus;
   }
   else {
      do nothing;
   }
}
```

Figure 3.6: Pseudo code for write operation in block i of the dual-key shift PQ.

and switches. For large time slots it is possible to insert the maximum $(RX + 2)$ packets into the shaper. Assuming that the shaper PQ can operate at 100 MHz (each operation takes 10 ns), Table 3:1 shows the maximum number of shaper operations possible for various link

link speed	1 byte	5 bytes	10 bytes	53 bytes	1500 bytes
100 Mbps	8	40	80	424	12000
500 Mbps	1	8	16	84	2400
1 Gbps	<1	4	8	42	1200

Table 3.1. Number of shaper operations possible per time slot at 100 Mhz (10 ns per operation).

speeds and time slots. As shown in the table above, for small time slots it might be necessary to limit the number of newly-arrived HOL packets serviced by the shaper. Any remaining packets will have to be serviced during the next time slot, potentially leading to the hold-back of packets which would otherwise have been transferred to the scheduler during the next time slot. The only solution to this problem is to carefully tailor the size of the time slot based on the link speed, minimum packet size, and *RX*. Non-HOL packets (other packets in the same flow are queued in the shaper) must also be processed by the shaper during the time slot. These packets are appended to the linked list corresponding to their flows without being added to the shaper PQ. Time required for this must also be taken into consideration when determining the minimum time slot.

3.4.3 Shaper-Scheduler Architecture and Operation

The overall architecture consists of the dual-key shift PQ used as the shaper PQ and a scheduler PQ. When packets first enter the system, their start and finish times are computed. After each time slot, new HOL packets are inserted into the shaper PQ, while at most two shaper-to-scheduler transfers need to be performed. During the first transfer the system time $V(t)$ is used as an input into the shaper PQ and a read operation is performed. The start time of the packet currently transmitting or just finished transmitting is used as input during the second shaper PQ read. If $V(t) < min(S(\text{ HOL packets queued in shaper PQ}))$, then both reads will return invalid results. In this case a shift operation is done, with the packet in the right-most block transferred to the scheduler and the system time set to the packet's start time.

Packets in the scheduler are serviced in increasing order of their finish times. Any PQ implementation with maximum operation time less than or equal to the time for the smallest packet to transmit can be used. An important parameter in deciding the implementation of the scheduler PQ is its maximum capacity. Ideally, this should be infinite. The RSE automatically achieves this by maintaining an array of linked lists, with each list indexed by the finish time of the packets in the list. For PQ architectures with a fixed capacity, careful analysis is required to determine the minimum PQ size. Otherwise, an eligible packet with a finish time smaller than that of the packet at the head of the scheduler queue might miss its deadline because of a full scheduler PQ.

The interface to the shaper-scheduler provides the following functions. Write into the shaper PQ, readout of an eligible packet in the shaper PQ, read of the top of the shaper PQ

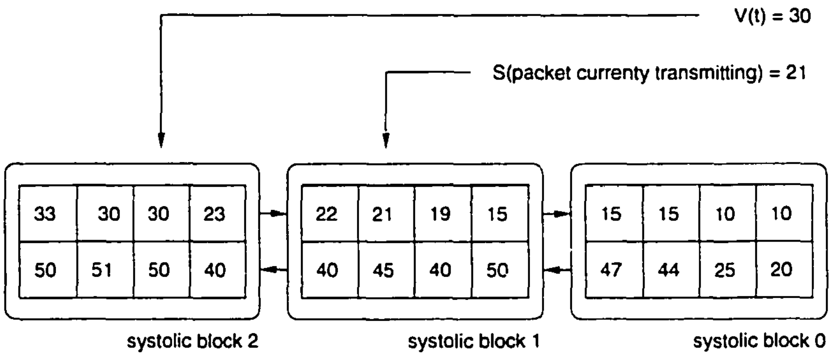


Figure 3.7: Tag arrangement in the hierarchical dual-key shift PQ.

without removing it (when there are no eligible packets), write into the scheduler PQ, and read out of the top of the scheduler PQ.

3.4.4 Implementation Issues

Despite the architectural simplicity, implementation issues limit performance for very large *N*. As seen in Figure 3.4 the input is distributed to all the blocks via a bus, while each block needs to drive the output bus. More blocks require a larger bus, which adds both costs for extra/larger buffers to drive the bus, and increases operation times due to the extra time required to drive the bus. Although not of the same magnitude, this problem is similar to that of the clock distribution problem in today's high-end microprocessors which use very expensive and complicated solutions to minimize clock skew throughout the chip. [48] proposes a systolic approach to isolate the size of the bus from *N* by dividing the shift PQ into (N/c) shift PQ systolic blocks of length *c* each. In order to maintain constant-time read of the highest priority tag for any given output link, the "at-least-1-entry-per-output-link" property is introduced. Each systolic block in the new architecture (includes the smaller length *c* shift PQ) retains atleast one tag for all possible output links. Since the number of output links is relatively small, $min(c) = RX$. However, for the shaper PQ $min(c) = max(W)$, which will not help with the bus loading problem. Also, the cost of maintaining *W* counters in each systolic block can be very expensive.

Without any modifications, the following situation, as shown in Figure 3.7, can occur. The tag with the requested start time will not necessarily be in the right-most systolic block.

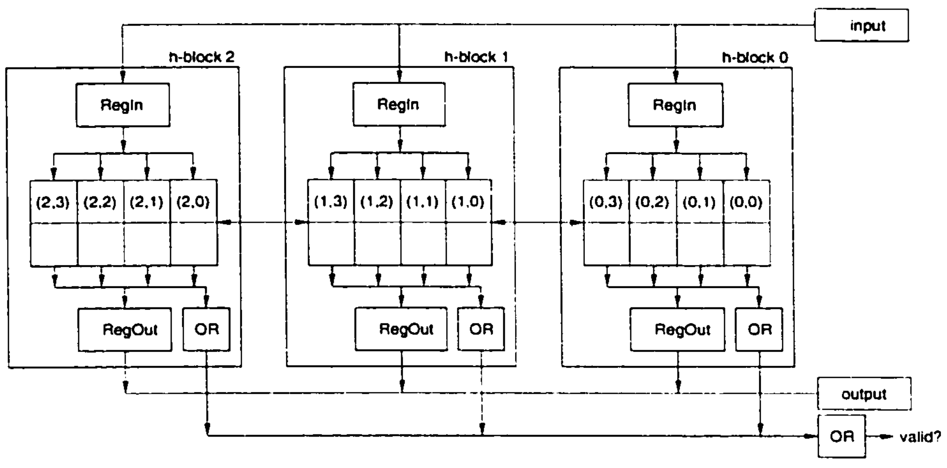


Figure 3.8: Block diagram of the hierarchical dual-key shift PQ.

If the first block does not have the requested tag, it will ask its left neighbor. In the worst case, the requested tag could be in the left-most block, meaning that each shaper read can take $(N/c) \times 2$ cycles for the result to get back to the right-most block. In order to maintain constant-time reads, we propose to treat each c-entry dual-key shift PQ as a block in a hierarchical dual-key shift PQ consisting of (N/c) blocks. This is shown in Figure 3.8. The inputs are still distributed to all the blocks, but with a major difference. Before, the input bus needed to drive the inputs of N blocks. In the new implementation the value on the input bus is latched into a register (RegIn) at each larger block, so it only needs to drive (N/c) inputs. Since the load on the bus depends on both the length of the bus and the number of inputs driven by the bus, this method significantly reduces implementation costs associated with large buffers. Similarly, during the read the output from each c-entry dual-key shift PQ is latched into a register (RegOut). The output of this register then drives the output bus of the hiearchical shift PQ. The savings in implementation costs results from smaller tri-state buffers for each shift block within each c-entry dual-key shift PQ. Instead of driving a bus which spans N blocks, these buffers only need to drive a bus spanning c blocks. After the read operation a shift operation is done by the block of c-entry dual-key shift PQs. The decision making process is shown in Figures 3.9 and 3.10. Since each hierarchical block (h-block) is able to make a decision based on just local information, the read, write, and shift operations still remain constant-time operations. However, the number of cycles

```
1..C1: latch RegIn(i);
2..C2: do read and latch RegOut(i);
3..if (RegOut(i) is valid) and (valid(i-1)==0)
4..   C3: valid(i) = 1;
5..   C3: drive do_shift_r_bus = 1
6..else if (RegOut(i) is valid) and (valid(i-1)==1)
7..   C3: valid(i) = 0;
8..   C3: re-insert RegOut(i) into block i's shift PQ;
9..else
10..   valid(i) = 0;
11..C3: do_shift_r(i) = do_shift_r_bus;
12..if (valid(i)==1)
13..   C4: (i,c) = (i+1,0);
14..   C4: output bus = RegOut(i);
15..if (do_shift_r(i)==1) and (S((i,0)) > S(RegIn(i)))
16..   C4: shift right;
17..        /*[(i,0),(i,1),...,(i,c)] =
[(i,1),(i,2),...,(i+1,0)]*/
```

Figure 3.9: Pseudo code for read operation in h-block *i* in the hierarchical dual-key shift PQ.

required for the read increases to 4 while 3 cycles are required for the write. The shift takes 2 cycles because of the latching of the control signals. In line 12 of Figure 3.9, h-block *i* has the desired tag, which has been shifted out of its own c-entry shift PQ, and latched into its RegOut. This leaves the entry (i,c) empty, which needs to latch the rightmost tag from h-block *i+1* (on its left). In lines 3-10, the condition where h-block *i* is valid, *i-1* is not, and *i-2* is valid, cannot occur because this condition implies that tags were not sorted properly based on their start times during the write (tags with the same start time must be in consecutive blocks). In line 2 of Figure 3.10, tempreg holds the tag in the leftmost block. If h-block *i* decides that it is the holding place for the new tag, then the previous leftmost tag needs to sent to be the rightmost block of h-block *(i+1)* for the left shift.

In the dual-key shift PQ, having to route *N* signals to the input port of the OR function makes implementing this impractical for a very large *N*. An alternate implementation is shown in Figure 3.11. The *valid_bus* is initially pre-charged to logical value 1. A block will output "1" if it drives the output bus, causing the charge in the output bus to discharge through the transistor (when the gate is set to "1" the transistor creates a short between the

```
C1: latch RegIn(i);
C1: tempreg(i) = (i,c);

if (S(RegIn(i)) > S((i,c))) {
  C2, C3: idle;
}
else if (S(RegIn(i)) == S((i,c))) and
         (F(RegIn(i)) >= F((i,c))) {
  C2: tempreg(i) = RegIn(i);
   C3: idle;
}
else if (S((i,c)) > S(RegIn(i)) > S((i,0))) or
        (S((i,c)) > (S(RegIn) >= S(i,c)) and (F(RegIn(i)) >= F((i,0))))) or
        (S((i,c)) == (S(RegIn) > S(i,c)) and (F(RegIn(i)) < F((i,c))))) {
   C2: insert RegIn(i) into block i's shift PQ;
   C3: idle;
}
else {
   C3: shift left;
/*[(i,0),(i,1),...,(i,c)] = [(i-1,tempreg(i-1)),(i,0),...,(i,c-1)]*/
}
```

Figure 3.10: Pseudo code for write operation in h-block *i* in the hierarchical dual-key shift PQ.

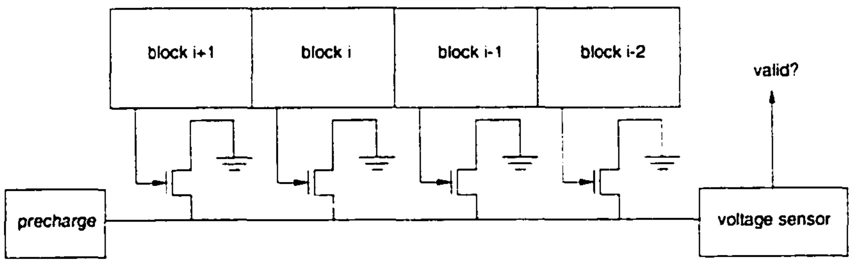


Figure 3.11: Determining validity of the output bus in the dual-key shift PQ.

source and drain). A voltage sensor at the other end will output a "1" if it senses a drop in voltage (valid) and "0" otherwise. The OR output of each h-block can then be used as the input into the OR gate for the hierarchical organization.

3.4.5 Evaluation

At an architectural level the dual-key shift PQ's implementation cost and performance scales well to N, M, and W. In terms of scalability, each additional flow simply adds another block to the PQ, while increasing M and W simply adds more bits to the encoded start and finish times. Increasing the maximum of these values increases implementation costs which increase by $O(W)$ and $O(M)$ respectively for the extra bits required to store these values. However, performance is affected by N due to the bus loading problem, while the operation time for the PQ increases at rate $O(W)$ and $O(M)$ due to the increased time required to compare a larger number of bits. Also, implementation costs can be very high for large N values due to the large buffers.

The hierarchical dual-key shift PQ isolates the bus load from N. However, the savings in buffers comes at a cost to performance. Instead of one cycle read and write operations, because the PQ is broken up into smaller pieces, extra steps need to be taken to maintain a consistent view throughout the entire PQ. As seen in Figures 3.9 and 3.10, a maximum of 4 cycles are required for each read and 3 cycles for the write operation.

In [48], maximum clock speeds of 40 MHz (25 nsec clock) are reported for this architecture. These results, however, were obtained from a 1.2 micron CMOS process. This is several generations old, and with today's submicron CMOS process, clock speed of over 100 MHz will be easily achievable. This is equivalent to 25 million shaper operations per second. Assuming a node with 16 input ports, and minimum packet size of 64 bytes, the shaper can support input link speeds of atleast 800 Mbps.

3.5 Sorted Constant-time One-Stage Architecture

This section presents a new architecture that does not employ the traditional two stage approach to shaper-schedulers. Two PQs, both of which sort on two keys, are used in tandem to provide the shaping and scheduling mechanism. The start PQ (SPQ) sorts packets based on start time and then finish time, while the finish PQ (FPQ) sorts in the reverse order, finish time and then start time. Instead of transferring eligible packets to the scheduler and then choosing a packet to transmit from those in the scheduler, the new architecture

SPQ

start time	10	10	10	7	6	5	4
finish time	35	30	20	14	30	15	25
flow id	17	5	13	10	9	3	7
	6	5	4	3	2	1	0

FPQ

finish time	35	30	30	25	20	15	14
start time	10	6	10	4	10	5	7
flow id	17	9	5	7	13	3	10
	6	5	4	3	2	1	0

Figure 3.12: HOL packet arrangement in one-stage architecture.

simply looks at all HOL packets before making a decision. For simplicity, we will first describe how the architecture works, before discussing details of the architecture.

3.5.1 Operation

A write operation simply inserts the new packet into both PQs. Only HOL packet tags need to be inserted. A packet which is not the HOL packet in a flow will not be considered for transmission until the HOL packet and all other packets that arrived before it have been transmitted. After a packet is chosen to be transmitted, the next packet in the flow, if any, is inserted into the dual bank of PQs.

A read operation only occurs after the transmission of a packet has finished. The result of the read is the next packet to be transmitted. This packet satisfies the following criteria. It must be eligible (its start time is less than or equal to the system time) and it must have the smallest finish time among all eligible HOL packets. As shown in Figure 3.12, we need to consider two cases. If the system time is less than the start time of the top block's tag in SPQ ($V(t)) \le S(SPQ(0))$), then no packets are eligible and no action is taken. Otherwise, the next packet to transmit is somewhere in FPQ. For example, if the system time is 6, then we see that there are three possible matches (blocks 1, 3, 5). At this point the desired tag resides in block 1. After the tag is removed from FPQ, it must also be removed from SPQ. The mechanisms used to remove this tag in constant time from both PQs are described in the next section. If there are more packets in the same flow, the next packet's start and finish times are computed and the corresponding tag is written into both PQs.

The interface of the one-stage shaper-scheduler provides the following functions. A readout of the packet with the smallest finish time among all eligible packets, a read of the top of the SPQ without removing it (when there are no eligible packets) and a write into the shaper-scheduler.

3.5.2 Start-Time PQ

In order to have a constant-time read operation, we extend the dual-key shift PQ proposed in Section 3.4.1. First, on a read operation the SPQ is required to remove the packet that was just scheduled (removed from the FPQ). Along with the start time, the flow id and finish time are broadcast on the input bus. Since only one packet per flow will reside in the SPQ, the logic for each block will be much simpler. It will drive the output bus if its contents match that of the input bus, and a shift right will occur starting at that block. The other blocks can decide whether to load in the block to its right (participate in the shift) or do nothing simply by comparing its contents to that of the input bus. Second, the SPQ does not require an output bus since only block 0's contents are needed during a shift operation. Both read and write operations require one cycle to complete. The write operation does not need any modifications.

Figure 3.12 shows the SPQ storing both start and finish times for each packet. Since only HOL packets are queued in both PQs, each packet can be uniquely identified by its flow id. This implies that the finish time does not need to be stored in the SPQ, since the SPQ only needs to output the smallest start time when there are no eligible packets in the FPQ. However, when we consider the following scenario we see that storing the finish time in the SPQ results in the saving of an operation. If based on the current system time there are no eligible packets to transmit, the system will either wait (a non-work-conserving server) or the system will set the system time to that of the smallest start time among currently queued packets (work-conserving server). If the finish time is not stored in the SPQ, three operations are required to determine the next packet to transmit in a work-conserving server. First, the SPQ needs to be accessed to read the smallest start time. After the system time has been updated, the FPQ then needs to be accessed to determine the packet with the smallest finish time. This will return the flow id of the packet, which is used to remove the corresponding entry from the SPQ. This is done because the smallest start time packet is not necessarily the one with the smallest finish time. However, if the SPQ also sorts packets

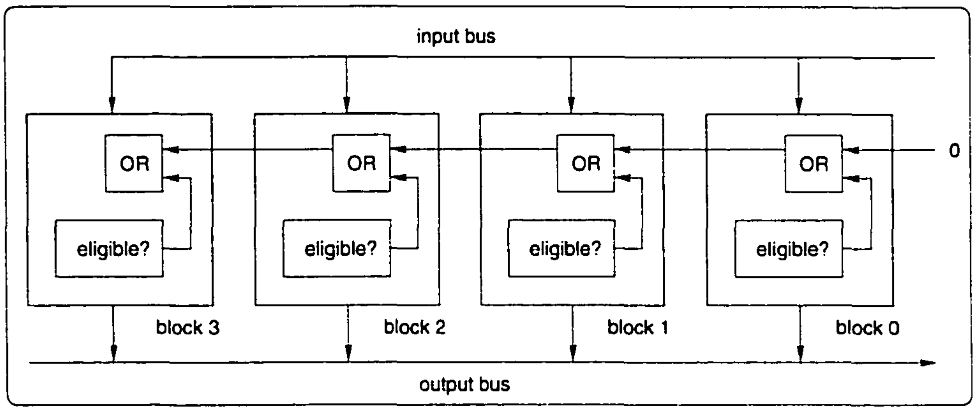


Figure 3.13: Block diagram of the finish-time PQ.

based on start and finish times, the first access to determine the smallest start time will also return the HOL packet with the smallest deadline. The second operation then uses the flow id to remove the corresponding packet from the FPQ. This results in a saving of one operation.

3.5.3 Finish-Time PQ

The dual-key shift PQ is extended to support the removal of the tag with the smallest finish time among tags with start time less than or equal to a specified time. As we saw in Figure 3.12 multiple non-contiguous blocks can have tags with start time less than or equal to the time on the input bus. Since tags are ordered in finish time order, the correct output is the right most match. However, the other blocks need to be notified that one of the blocks on its right will be driving the output bus. As shown in Figure 3.13 a chain of OR gates are used. A block will only drive the output bus if its tag is eligible and the output of the OR gate from the block to its right is 0 (not eligible). It is easy to see that only one block will drive the output bus, and will always be driven during a read since the FPQ will only be read if there is an eligible packet (decided by reading SPQ).

Due to the propagation delays in the OR gates, an eligible packet could reside in the left-most block and the right-most block. The output of the right-most block's OR gate will need to propagate through all the blocks in the PQ before the left-most block is notified that it shouldn't drive the output bus. Coupled with the bus loading problem on the input and

```
C1: latch RegIn(i);
C2: do read and latch RegOut(i);
if (RegOut(i) is valid) and (OR(i-1)==0) {
    C3: OR(i) = 1; valid(i) = 1;
    C3: drive do_shift_r_bus = 1;
}
else if (RegOut(i) is valid) and (OR(i-1)==1) {
    C3: OR(i) = 1;
    C3: re-insert RegOut(i) into block i's shift PQ;
}
else {
    C3: OR(i) = 0; valid(i) = 0;
}
C3: do_shift_r(i) = do_shift_r_bus;
if (valid(i)==1) {
    C4: (i,c) = (i+1,0);
    C4: output bus = RegOut(i);
}
if (do_shift_r(i)==1) and (OR(i-1)==1) and (OR(i)==0) {
    C4: shift right;
        /*[(i,0),(i,1),...,(i,c)] = [(i,1),(i,2),...,(i+1,0)]*/
}
```

Figure 3.14: Pseudo code for read operation in h-block *i* of the hiearchical finish-time PQ.

output buses, the implementation costs and performance degradation limit the number of blocks in the PQ. Similar to the solution proposed in Section 3.4.4, the N-entry FPQ is broken up into a hierarchical organization of c-entry FPQs. The block diagram is the same as the one shown in Figure 3.8, except that a chain of OR gates connect the N/c c-entry h-blocks. The control for each h-block's read operation is shown in Figure 3.14. The write operation remains the same, except the roles of the start time and finish time are reversed. This hierarchical organization reduces the bus loading, and decreases the maximum number of OR gates a signal must propagate through. This, however, comes at a cost to performance as the time required for each read operation increases from one to four. To increase the performance of each c-entry FPQ block, a scheme similar to that of carry-look-ahead used in carry-ripple adders can be used to reduce and bound the maximum propagation delay of the OR gate chain.

3.5.4 Evaluation

As we will see in Chapter 5, the one-stage architecture provides the same functionality as the two-stage architectures. The main advantage of this architecture is that it eliminates the need for the shaper to scheduler transfer of eligible packets. Scalability with respect to *N*, *W*, and *M* are the same as the two-stage architecture proposed in Section 3.4. Using a hierarchical organization can control performance degradation for large *N*.

3.6 Summary

In this chapter we first proposed two new architectures for queueing of packets and determining eligibility. For the first architecture, we proposed a new shaper PQ which can find the eligible packet with the smallest finish time in constant time. The second architecture eliminates the need for the shaper-to-scheduler transfer of eligible packets by using a unique one-stage solution. This is a deviation from the traditional, and more intuitive, two-stage solution that has been proposed. We evaluated these two new architectures at an architectural level and also discussed implementation issues and proposed changes to the architecture to minimize some of the implementation-related scaling effects (i.e., bus loading, multiple gate delays).

We showed that the two new hardware shaper-scheduler architectures provide the bare essential tag transfer mechanisms needed for any mix of traffic shaping and link scheduling algorithms. Both offer good performance and are easy to implement, and hence can be used in guaranteeing QoS requirements in high-speed networks.

CHAPTER 4

IMPLEMENTING TRAFFIC SHAPING AND LINK SCHEDULING ON AN END-HOST SERVER

4.1 Introduction

This chapter examines the implementation of traffic shaping and link scheduling mechanisms on an end-host server. Delivering QoS guarantees requires an end-to-end solution [3]. In other words, traffic shaping and link scheduling mechanisms must be in place both within the network at switches and routers, and also at the flow's source (i.e., the server). Although most research focuses on implementation at nodes within the network, this chapter focuses on implementing traffic shaping and link scheduling on an end-host server with a single outgoing link. During heavy loads when a large number of outgoing flows are being set up by the server, congestion at the outgoing link requires traffic shaping and link scheduling to guarantee that each flow meets its QoS guarantees. Unlike switches and routers which simply react to incoming packets, depending on how the shaper-scheduler is implemented on the server, the resulting implementation will either react to packets generated by applications running on the server, or will directly impact the generation of packets by applications. This direct impact can change the QoS received by each flow. To highlight these issues, and to motivate the need for a dedicated shaper-scheduler on the server, we will present and examine more traditional implementation strategies. By traditional we refer to software implementations, either at the operating system or application level, which run on the same server CPU(s).

This chapter presents a network interface architecture with dedicated traffic shaping and link scheduling support. When a new flow is initiated, its traffic parameters are downloaded into the network interface through its programmable interface. Assuming that the application receives enough CPU to process its stream into packets, the network interface will smooth the flow's stream, regardless of how bursty the actual processing packet is, and schedule transmission on the outgoing link so that all flows receive their desired QoS. This approach allows the shaper-scheduler to operate concurrently with the rest of the server, thereby reacting to the flows. The architecture also significantly reduces server CPU load during heavy loads (both CPU usage and number of flows being served). Very fine-grain link multiplexing, and consequently more diverse set of QoS guarantees, can be easily supported in this architecture for a large number of flows.

The rest of the chapter is organized as follows. A detailed examination of shaper-scheduler implementation issues on end-host servers is given in Section 4.2. In Section 4.3 we present our network interface architecture, and describe its operation within the framework of a streaming server. We also present two possible implementations of our architecture. Section 4.4 presents a performance evaluation of these two implementations, along with a brief description of our simulation environment and traffic models. Section 4.5 concludes the paper with a summary of our work and a brief list of future directions.

4.2 Implementation Strategies

This section examines several traditional means of implementing a shaper-scheduler on an end-host server. We will discuss the mechanisms and limitations of these solutions. We first state assumptions regarding the server configuration.

4.2.1 Assumptions

Figure 4.1 shows a generic server that we assume in our work. One or more processors are connected to secondary storage devices and the network interface through an interconnect. In most PC server configurations, this is just a shared bus. However, as processors and network link speeds become faster, and larger storage devices become more readily available, servers will be capable of serving more and more requests. Based on this trend, the current shared bus will become a major bottleneck, and will be replaced by a faster and more effi-

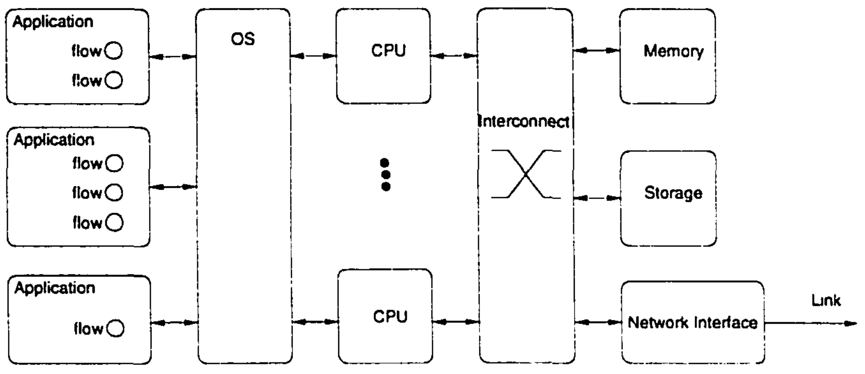


Figure 4.1: A generic server model.

cient interconnect mechanism. Although we don't assume a specific type of interconnect, we do assume that the interconnect will be fast enough not to be the bottleneck.

We also assume that applications will receive enough CPU cycles for packet processing [1]. This implies a process scheduler on the OS, which is important for applications which need to process and transmit data at periodic intervals. Without this any shaper-scheduler implementation will be made ineffective. We assume that each application is associated with one or more flows, and that each flow is specified by a set of traffic parameters (i.e., burst size and minimum burst interval). For example, a stored video-on-demand server application can transmit several different flows (i.e., video streams) to remote clients for real-time playback. Although the entire video file is stored on disk, streaming the file allows the user to start playback without waiting for the entire file to download. As video files tend to be very large, streaming results in more efficient use of network resources, and allows the server to deliver good performance to a larger number of clients. In this example, each video stream's traffic parameter is specified by its playback rate. Transport layer protocols such as TCP simply try to detect and avoid congestion within the network, and are not suitable for such streaming applications which require a specified spacing between each transmission burst.

4.2.2 Application-level traffic shaping

Applications such as streaming servers which need to pace each flow at a specified rate, can take advantage of a traffic shaper. However, without any OS-level support, one solution is to have each application pace its own flows. This assumes either a FCFS or other link scheduler at the transmit queue. Such self-pacing requires that each application monitor each flow and hold back transmissions for a flow if it violates the flow's traffic parameters. Since a send call to the OS will attempt to transmit the data as soon as possible, the application can do one of two things to pace out its data transmission. It can either process the data and then perform a check (conformance to the traffic parameters) right before the send call, or it can delay the processing of the data (using timers) to the next start time. Both methods require a form of delay mechanism, which is not feasible nor accurate for small delays and large number of applications and flows. If each application or flow is processed by a different process or thread, the number of context switches from user to user and from the user to kernel during the send call can add significant overhead and load on the server CPU(s). For example, if each flow transmits 2KB each 0.1 second, 1000 simultaneous flows will require at least 10,000 task switches. To reduce this overhead, each application can increase both the burst size and burst interval to 20KB per second. This, however, requires larger buffers at each node in the network for each flow. Larger bursts also can increase the probability of packet drops during congestion within the network.

This type of self-shaping mechanism is currently employed by current commercial stream servers such as Apple's QuickTime Streaming Server, Microsoft's Windows Media Services, and RealNetwork's RealServer. Under low server CPU loads, such self-time mechanisms might function well. However, under heavy loads, especially when the streaming server application must share the server CPU with other applications, the amount of time the streaming server application must wait for the server CPU could be much larger than a flow's minimum burst interval. This results in interrupted playback at the client.

4.2.3 Operating system support

An alternative solution to self-shaping, is to add shaping and link scheduling support at the OS level. Whenever the application gets access to the server CPU, each flow can buffer enough data at the shaper to last several burst intervals, and allow the OS to perform the traffic shaping and link scheduling. This implies that the actual send call is not made until the scheduler decides to transmit an eligible packet. Also, to ensure that flows are shaped according to their parameters, the shaper needs to be periodically executed. Although in Section 3.3 we defined this interval to be after each packet transmission, an actual implementation can make this period larger. This allows the shaper-scheduler to batch schedule a number of packet for transmission which become eligible during each period. However, a large period can distort a flow so that it no longer adheres to its traffic parameters. Also, as we will show later, the amount of processing required to shape and schedule is not trivial. This load grows with the number of flows, thus taking away CPU from applications, and lowering the number of flows supported by the server. In the worst case, the processing of the shaper and scheduler could interfere with the processing required by the applications.

4.3 Dedicated NIC Support

Both self-shaping and OS-supported shaping and scheduling rely on running on the same CPU as the applications. When the number of flows and load on the server CPU are low, both these schemes will work well. However, when either increases the processing needs of the shaper-scheduler will interfere with the processing needs of the applications. Our solution is to add dedicated support on the network interface (NIC). This allows concurrent operation of the shaping and scheduling with the rest of the server. The following subsections describe the basic operation of our NIC, and two possible implementations.

4.3.1 Basics

Flow setup

An application will setup a new flow only if an admission control algorithm [3] determines that the resources required to process and transmit the new flow will not exceed server resources. The admission control algorithm also guarantees that flows that are currently in service will not be affected when the new flow is admitted. Once the flow is admitted, an internal flow id is assigned to that flow. All references to the flow are made using the flow's flow id. The flow's data parameters are then downloaded into the NIC's shaper-scheduler.

Packet movement

Before data can be transmitted it needs to be packetized and moved to the NIC buffers. Assuming a UDP-like protocol, each flow's bursts will be processed by the UDP and IP protocol stack in the OS, before being presented to the device layer. At the device layer (e.g., ethernet), an ethernet header is attached and then copied to the NIC. In order to reduce the packet processing and movement overheads [5] [20] [21] [24] [49] [57] [63] [65], many researchers have proposed various network interface architectures. Single copy schemes [5] [20] [21] [63], which move data directly from user space to the NIC buffers, attempt to reduce the multiple memory copies necessary as the data moves down the various protocol stacks. Other schemes move some or all of the packet processing onto the NIC [23] [49] [77]. Instead of adding to the NIC, other schemes introduce new-buffer management mechanisms [25] [26], reduce DMA overheads [9], or give user-level applications direct access to the NIC [27] [51] [82].

We borrow from these results by assuming the following model. Each flow is assigned a user-level memory space which does not get paged out of physical memory. At most, the amount of memory will be enough for 1 or 2 seconds worth of data. The address and size of this memory will be fixed for the duration of the flow. The NIC is initialized with the starting address of this block of memory at the time of flow setup. For simplicity, we assume that all addresses are physical addresses and no address translation is needed for virtual addresses. Packets are then copied into NIC buffers using a DMA mechanism on the NIC. At this point, the packet has been fully processed and is ready for transmission, requiring no further packet processing at the NIC.

Buffer management

To amortize DMA overheads, the buffer manager on the NIC downloads several packets at a time for each flow. Each flow will have an associated linked list of packets in the NIC buffers. The buffer manager keeps track of the current number of packets in the buffers for each flow, and downloads packets from memory when needed. When the scheduler determines the next packet to transmit, the packet's flow id is used by the buffer manager to read the packet from NIC buffers into the NIC's physical interface FIFO. A separate linked list

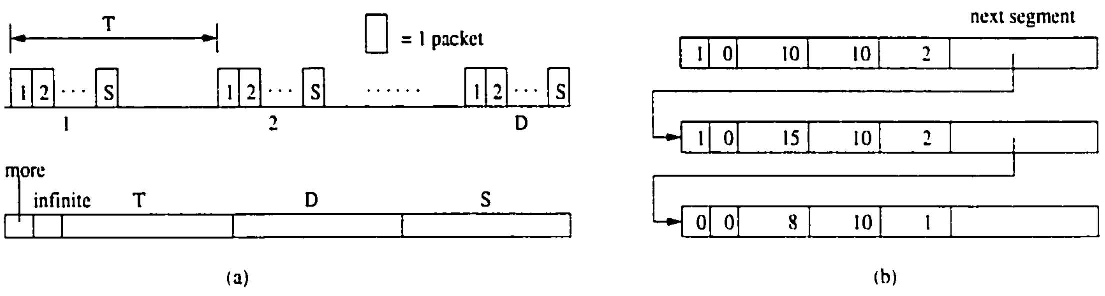


Figure 4.2: Data structure used to encode traffic parameters.

is used for best-effort packets. Best-effort packets are transmitted by applications which are not associated with any flows and do not require any QoS guarantees.

Control interface

A memory-mapped control interface is used to communicate with the NIC. All commands (download traffic parameters, stop a flow) are delivered through the command FIFO. Any exceptions, status information, and shaper-scheduler requests are made through a request FIFO.

Traffic parameters

Figure 4.2(a) shows the data structure (schedule segment) used to encode the traffic parameters for each flow. Like most shaping and scheduling disciplines, we use a simple rate-based scheme. The rate is specified by its burst interval (T), burst size (S), and duration (D). The duration refers to the number of bursts required to send the entire flow. For a stored video file, this is simply the file size divided by S. For a continuous flow (live video, audio), the infinite flag is set to tell the shaper-scheduler to ignore D. Both T and S are in units of chunks, which are fixed-size segments. We assume that all packets are integer multiples of chunks. For example, if S=2, and T=20, then the flow requires 2 chucks to be transmitted for every 20 chunks transmitted.

This traffic model can also be used to characterize VBR-encoded video streams. Smoothing techniques [28] [29] [61] [64] can be used to compute a transmission schedule which consists of a small number of constant-rate transmission intervals. This can be encoded as several schedule segments, as shown in Figure 4.2(b), which uses the more flag to chain 3 schedule segments.

Shaper-scheduler operation

Each active flow has an associated shaper tag which is queued in the shaper queue. The shaper tag consists of the flow id, the start time, and finish time of the flow's HOL packet. The start time is the finish time of the previous packet plus the burst interval (T). For the first packet in the flow, its start time is the current time. An internal counter keeps track of the time by counting the number of chunks scheduled for transmission so far. The finish time is the packet's start time plus T. In other words, its deadline is the earliest time the next packet in the flow can begin transmission. When the HOL packet becomes eligible (its start time is greater than or equal to the current time), the shaper will create a scheduler tag (flow id and finish time) and insert it into the scheduler queue. If there are more packets to transmit in the flow, the shaper reads the flow's schedule segment to create a new shaper tag. If this packet is the last in the flow, an end-of-flow message is written into the request fifo. After each packet transmission, the shaper will move any packets that have become eligible, while the scheduler determines the next packet to transmit. The scheduler then writes a data tag (flow id) into the transmit FIFO, which is processed by the buffer manager.

4.3.2 Hardware Implementation

Figure 4.3 shows the block diagram of the hardware implementation, which consists of two main blocks. The buffer manager contains the packet buffers, as well as the state machine to update the linked list of packets and interact with the DMA engine. The shaping and scheduling is performed by the control block, which is shown in Figure 4.4. A pointer memory, indexed by flow id, keeps track of the list of scheduler segments for each flow. Each line of the schedule segment SRAM contains one schedule segment. Since a flow can have multiple schedule segments, a FIFO (not shown) keeps track of free lines in the schedule segment SRAM that can be used to store new schedule segments. When a flow ends, or when the schedule segment ends, its address is returned to this FIFO. The heart of the shaper and scheduler is a dedicated priority queue mechanism [13][48], which provides

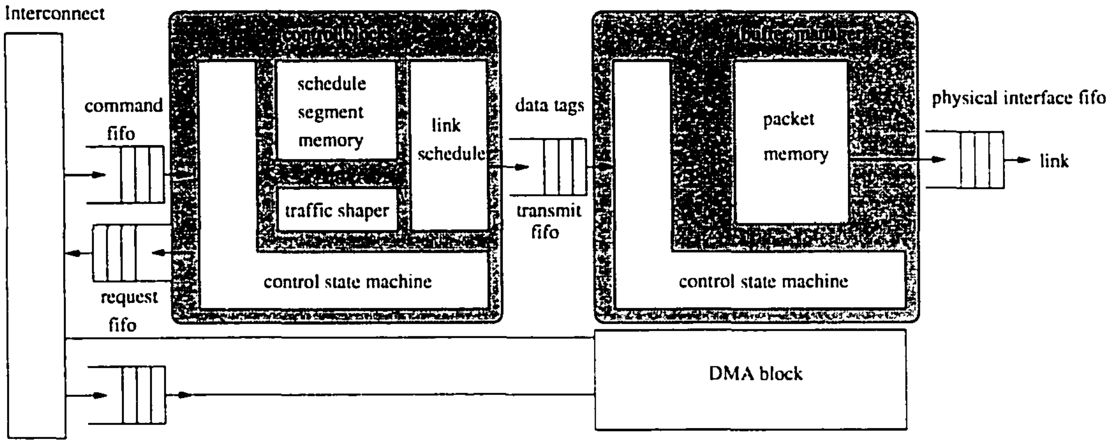


Figure 4.3: Block diagram of hardware implementation.

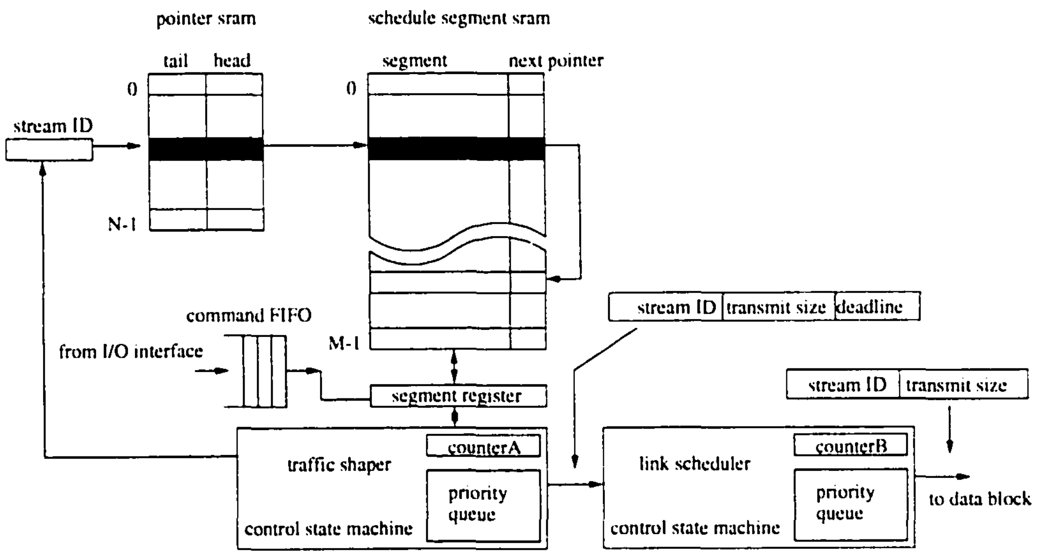


Figure 4.4: Block diagram of the control block.

constant time queueing, sorting, insertion, and removal of tags. The shaper's priority queue sorts tags based on the start time, while the scheduler's priority queue sorts tags based on their finish time.

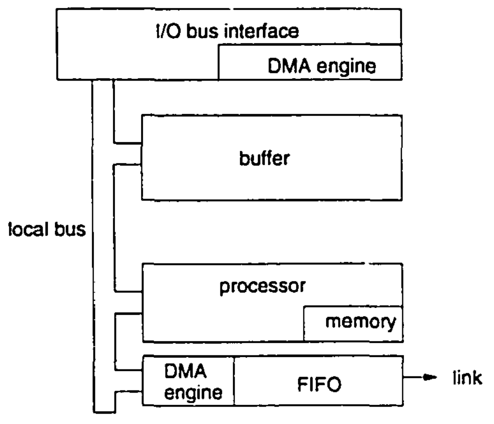


Figure 4.5: NIC with a dedicated processor.

4.3.3 Software Implementation

The NIC architecture for the software implementation is shown in Figure 4.5. This is similar to Myrinet's network interface [10] which uses the LANai processor and three DMA engines to move data to and from the network interface. Whereas Myrinet uses a byte wide Myrinet physical connections, our architecture is not dependent on the physical interface and can be built upon any link technology.

All schedule segments, packets, and other information are stored in onboard memory which is accessible by the server processor. The DMA engine and transmission FIFO are addressable by the NI processor. We also assume that the transmission FIFO has a DMA state machine which allows it to read out packets from memory without involving the NIC processor.

Figure 4.6 shows pseudo code for the software implementation. Note that the buffer management functions are not shown. The command interface is simply a region of the NIC processor's memory which can be accessed by the server processor. Flow setup and new schedule segments are written here. The NIC processor polls this memory and updates the schedule segment list. As with the hardware implementation, each active flow has a linked list of schedule segments associated with it.

```
main() {
  while 1 {
  if top of scheduler pq is eligible for xmit
    create and insert a data tag;
    remove top entry and sort scheduler pq;
  else if data tag FIFO not full and best-effort packets have been released by shaper
    insert a best-effort data tag;
  else if top of shaper pq is eligible for release and scheduler pq is not full
    remove top entry from shaper pq;
    if necessary
      create and insert new entry for shaper pq;
      sort shaper pq;
      create and insert entry into scheduler pq;
      sort scheduler pq;
    update schedule segment;
  else if necessary
    release a best effort packet;
  else if command FIFO not empty
    process command FIFO;
endwhile
endmain
```

Figure 4.6: Pseudo code of the software implementation.

As with the hardware implementation, the traffic shaper will insert as many entries into the link scheduler as possible. This is done to prevent missed deadlines. Consider two connections with the same eligibility time. If only a single entry is written into the scheduler PQ before running the link scheduler, the stream with the less urgent deadline can be scheduled and transmitted first, causing the other stream to miss its deadline.

Instead of transferring the actual page into the transmission FIFO, the NIC processor sets up the DMA engine on the FIFO. Similarly, when reading a page from source into NIC buffers, the NIC processor sets up the DMA engine on the I/O bus interface. Using DMA engines frees up processor cycles for other computations. Since the NIC processor is not actively involved in the actual transfer of packet data, a notification scheme is used to signal the NIC processor at the end of a page write into NIC buffers and at the end of a page transmission. This is needed to manage the linked list of pages and to keep track of the NIC buffer usage.

4.4 Evaluation

To evaluate our proposed architecture we developed a simple event simulator using C, and modeled both hardware and software version, also in C. This provides us with a common framework which makes comparing our results more meaningful. This also allows us to use the same traces and setup configurations to evaluate both implementations. Simulation results show the efficacy of our architecture in providing QoS-sensitive link multiplexing, especially when dealing with a large number of streams with widely-varying rates on a very high capacity link. We introduce the concept of period division and show the performance improvements obtained by using such fine-grain link multiplexing, and how our architecture can support this feature.

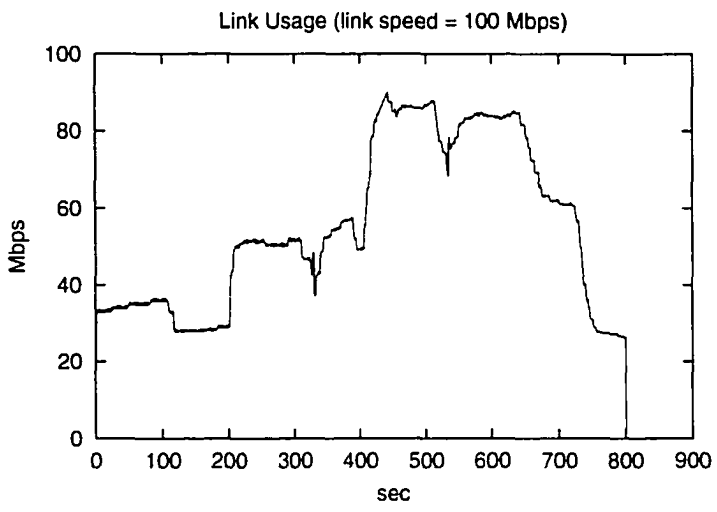
4.4.1 Simulation Environment

To evaluate our implementations, we wrote a simulator which enabled us to model and simulate both hardware and software components on a single platform. This simulator is presented in Chapter 6. Combining event-driven simulation techniques and software performance estimation, we were able to simulate long periods of time using our compiled simulator. By adding delay statements into the code for software components, we were able to avoid the issue of modelling a specific processor for the software implementation.

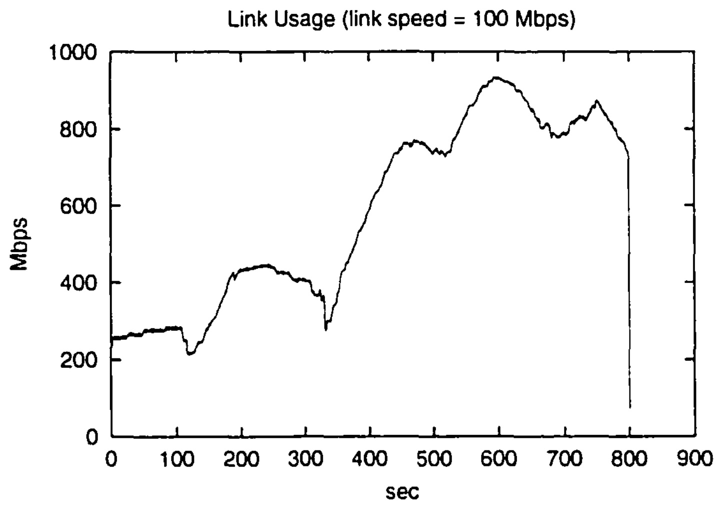
4.4.2 Reducing Delay and Delay Jitter

In our simulations we used link speeds of 100 Mbps, 622 Mbps, and 1 Gbps, with a fixed packet size of 128 bytes. Streams that need to burst data in larger sized packets can easily do so by using multiples of 128 bytes. By parsing the packets on the outgoing link based on their stream id, we were able to measure the delay jitter seen by the end clients.

At 100 Mbps the number of simultaneous streams ranges from 20 to 140, while the total number of streams during each simulation run was close to 200. For the 1 Gbps link, these numbers were between 120 to 580, and over 1000 streams. To quantify overall performance we measured the average delay and delay jitter for each stream over a one second interval. We then converted the delay jitter number into a percentage value (average deviation)



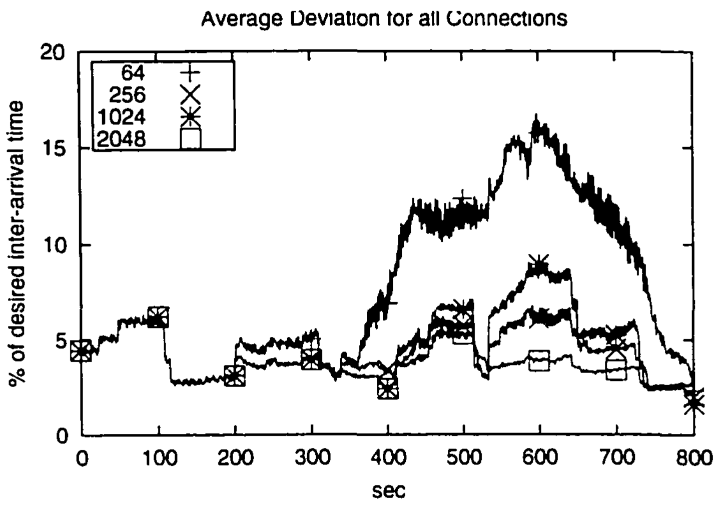
(a) 100 Mbps



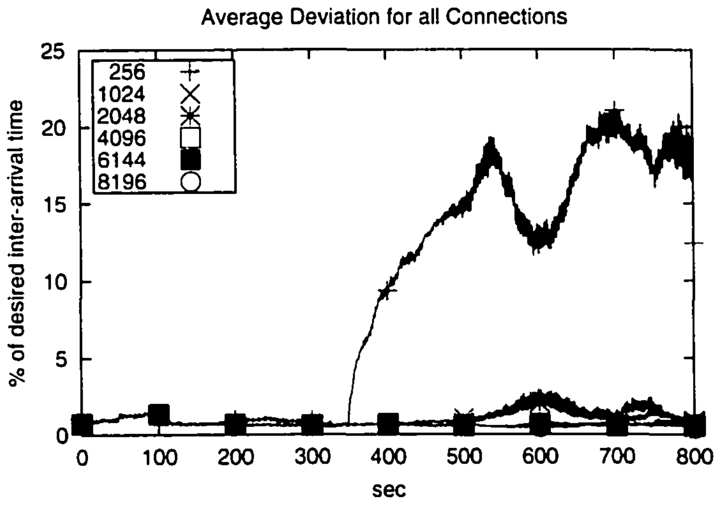
(b) 1 Gbps

Figure 4.7: Link utilization at 100 Mbps and 1 Gbps link speeds.

based on the desired, or requested, delay. For example, a stream requesting data transmits every 1 msec will see 10% delay jitter if transmissions occur every (1 ± 0.1 msec). These values were then averaged across all streams for each 1 second interval. Figure 4.7 shows the measured link utilization using a 100 Mbps and 1 Gbps link. The graph for the 622 Mbps run has the same profile in that utilization starts out very low and nears full capacity towards the end of the simulation run. Figure 4.8 shows average deviation values. As



(a) 100 Mbps



(b) 1 Gbps

Figure 4.8: Performance measurements at 100 Mbps and 1 Gbps link speeds.

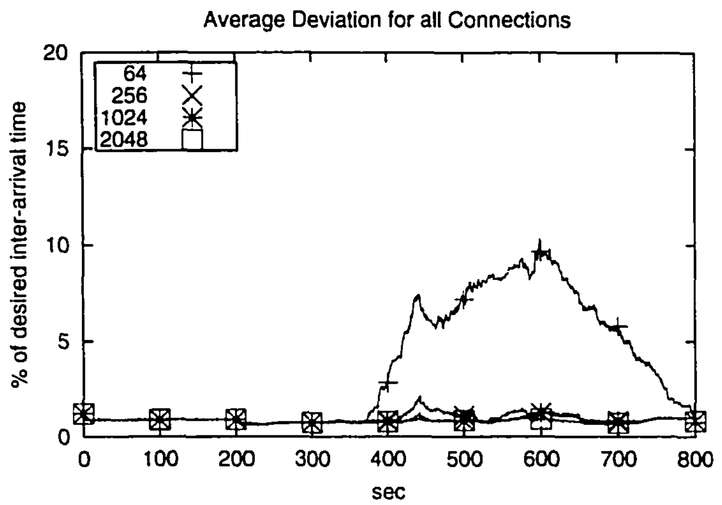
expected, the average deviation increases with increased load on the network link. By increasing the size of the scheduler PQ, we can force the traffic shaper and link scheduler to look ahead even further in time to determine a better link schedule. As shown in the graphs deviation values remain constant throughout the simulation run for very large scheduler PQ sizes. We also see that after a certain size there is no further drop in deviation despite further increases in the scheduler PQ size.

This is mainly because the rates differ greatly among the various streams. Some of the video streams transmit large amounts of data (5 to 26 KB) every 33 msec, while other streams transmit much smaller amounts of data (100 to 1000 bytes) every 1 to 2 msecs. As a consequence these smaller period streams can potentially wait past their deadlines if they get queued behind several large bursts. Even small jitter values translate into large percentage values because of the relatively small period sizes. Increasing the scheduler PQ size can reduce deviation only up to a certain point because doing so does not solve the problem where the smaller period stream gets queued behind several large bursts.

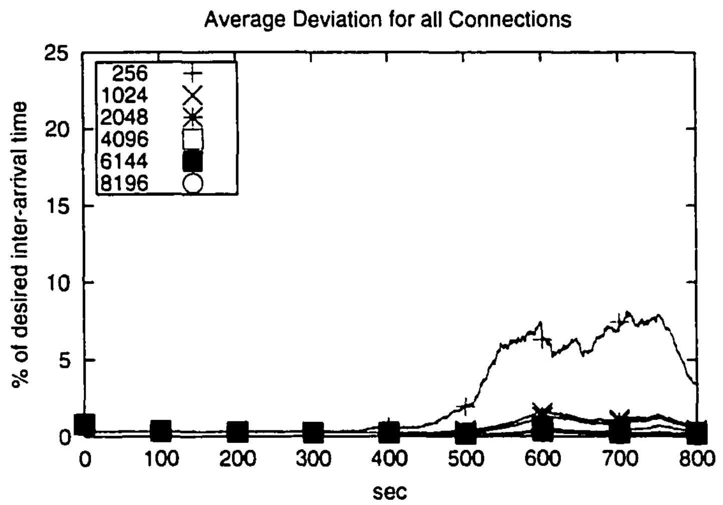
By reducing the size of the large bursts to match the burst size of the smaller period streams we can significantly reduce deviation of these smaller period streams. For example, if a video stream needs to burst 25 KB every 33 msecs, we can divide the period such that bursts are reduced to 750 bytes every 1 msec (factor of 33) or even 1.9 KB every 2.2 msec (factor of 15). We refer to this process as *period division*. Since all streams now have similar burst sizes, the same blocking problem discussed in the previous paragraphs results in a much smaller queueing time for the smaller period stream. As seen in Figure 4.9 deviation is reduced compared to the deviation values without the period division and using the same scheduler PQ size. This last observation is particularly important for the hardware implementation due to the hardware costs in building very large PQs. This is explained in the next section. Figure 4.10 shows deviation values for one of the smaller period streams. As expected, these streams gain the most by period division. From Figure 4.10(b) we see that the same deviation can be achieved with a 256 size PQ using period division or 8196 size PQ without period division, resulting in a factor of 32 in hardware savings. We can see the same trends in Figure 4.10(a).

4.4.3 CPU Load

Figure 4.11 shows processor loads for some of the simulation runs. Since the priority queue and its operations are implemented as a binary heap, processor load does not increase significantly with increased PQ size. However, load does increase by a factor of 2 to 4 when we use period division because the number of operations to transmit the same amount of data has increased. At low link speeds the load is below 10%, but approaches 100% at high



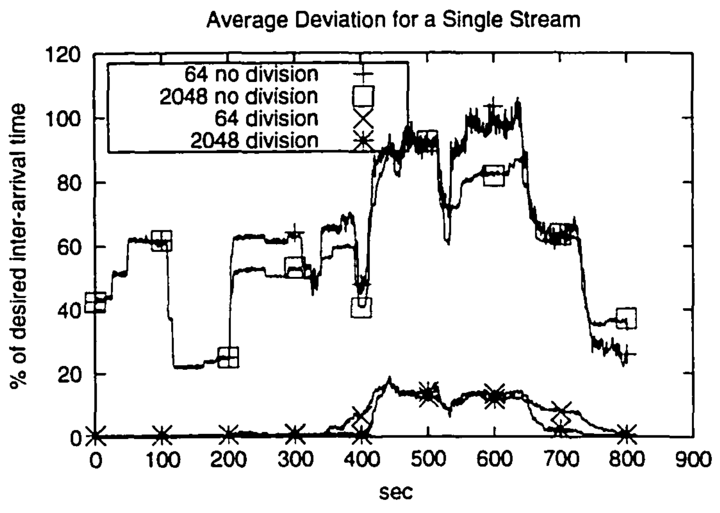
(a) 100 Mbps



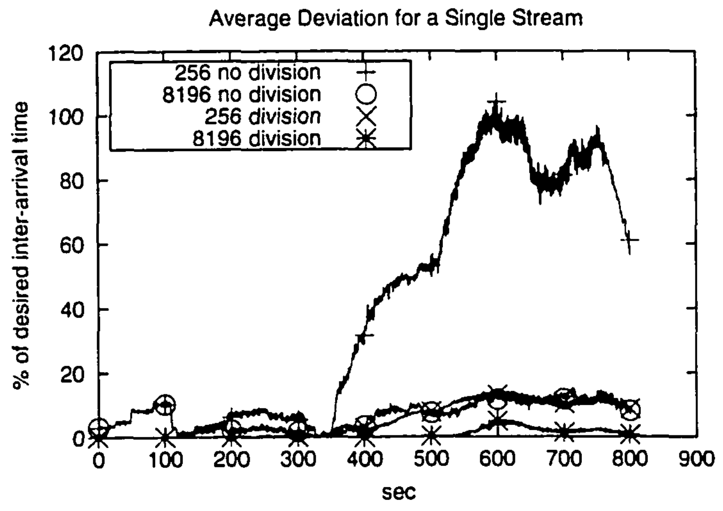
(b) 1 Gbps

Figure 4.9: Performance measurements after incorporating period division.

link speeds. For even higher link speeds and larger number of simultaneous streams, the computing load required will exceed the capacity of the processor. This means that the server will have to reduce the number of simultaneous clients to deliver the same level of QoS across all streams. Otherwise, the shaper-scheduler's load will exceed 100%, causing the link to go idle even when there are packets to transmit.



(a) 100 Mbps

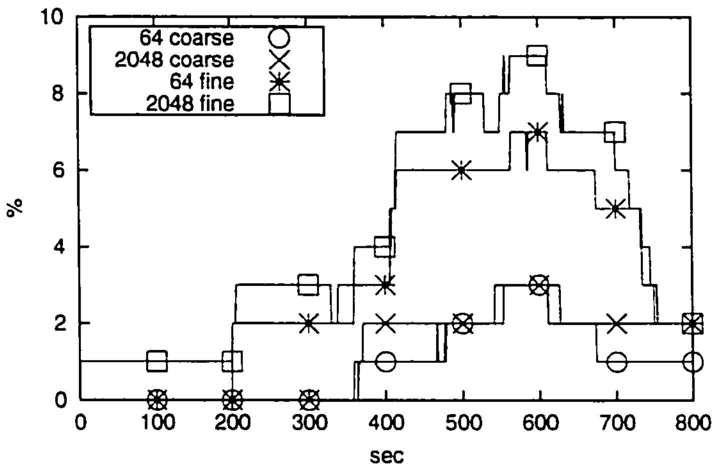


(b) 1 Gbps

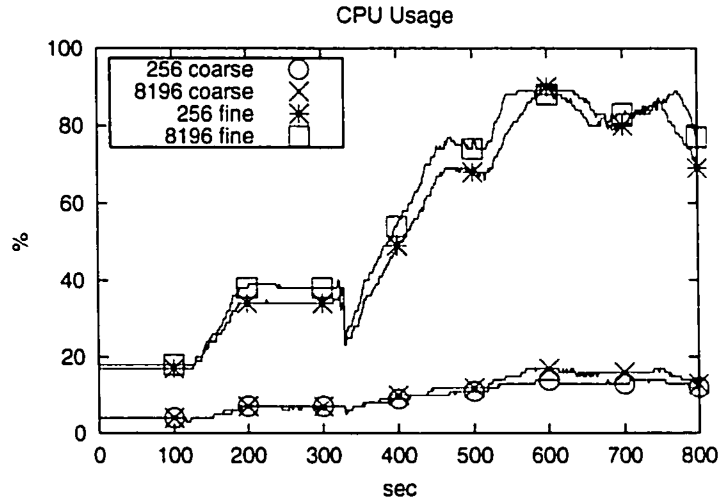
Figure 4.10: Performance measurements of a single small period stream.

4.5 Summary

In this chapter we proposed and evaluated a network interface architecture with dedicated support for QoS-sensitive transmission. We defined an architecture and low-level functions for supporting traffic shaping and link scheduling. Based on hardware and software imple-



(a) 100 Mbps



(b) 1 Gbps

Figure 4.11: Processor loads.

mentations we measured performance seen by the user in terms of delay and jitter, and showed the effect of increasing the scheduler PQ size in reducing delay jitter. We also showed the effectiveness of fine-grain link scheduling in significantly reducing delay jitter without using very large capacity scheduler PQs, which significantly lowers hardware implementation cost. We showed that, by moving the implementation into the network interface, the server can take advantage of the concurrent execution of operations. This allows the server to support finer levels of QoS, without burdening the server processor.

Not only does this free the server processor to process other tasks, but it also results in a larger number of connections receiving their desired QoS. This allows the system to make the best use of server resources in terms of processor time and link bandwidth.

CHAPTER 5

A PROGRAMMABLE TRAFFIC SHAPING AND LINK SCHEDULING ENGINE

5.1 Introduction

The focus of shaper-scheduler architecture work in Chapter 3 and in the current literature [14] [58] [66] [67] [70] [87] has been mainly on efficient mechanisms for queueing HOL packets, determining eligible packets, and sorting eligible packets based on their finish times. However, an important issue that is largely ignored is the implementation of the controller and associated datapath. The controller is responsible for queueing newly-arrived packets, maintaining flow state, computing start and finish times, and keeping track of the system time. The datapath is characterized by the connection of the various state machines, memory elements, and the shaper and scheduler queues, and determines the efficiency of data transfers. Consequently, the overall performance of the shaper-scheduler will depend on the efficiency of the controller and datapath implementation. Previous related work has either not addressed this issue or has simply assumed a generic processor block with no further details. Although hard-wiring an implementation for a specific algorithm can provide the best performance, it provides the least flexibility and will require a re-design for other algorithms. To leverage on the high performance of hardware, and yet provide enough flexibility, we propose a shaper-scheduler processing (SSP) engine.

The SSP is a very simple micro-controller-based processor, with separate downloadable program memory and data memory. Based on a small instruction set, the processor can be programmed to execute any mix of traffic shaping and link scheduling algorithm. The shaper-scheduler architectures from Chapter 3 are incorporated into the datapath as memory-mapped devices. Since these devices provide the queueing, sorting, and searching functions, any algorithm can be quickly programmed by considering just the timestamp computations.

The rest of this chapter is organized as follows. In Section 5.2 we first detail several shaper and scheduling algorithms discussed in Section 3.2, focusing on the timestamp computation aspects of each. Based on this we propose a generic processing unit and datapath implementation. An overview of the operations and datapath are given in Section 5.3. In Section 5.4 we describe a memory organization that can store the data structures required for any shaping and scheduling algorithm implementation. Section 5.5 describes the processor. An evaluation of the SSP is given in Section 5.6. We then show how this can be used to effectively implement a few well-known algorithms.

5.2 Algorithms

5.2.1 Leaky-bucket Shaper

The shaper generates tokens for a flow i at a rate of ρ_i, where σ_i is the maximum number of tokens than can be accumulated in the bucket. Assuming the k^{th} packet from flow i arrives at time $A(p_i^k)$, and requires L_i^k tokens we can calculate the packet's start time $S(p_i^k)$ (earliest time the packet is eligible) as shown in Figure 5.1(a). The algorithm simplifies to Figure 5.1(b) when dealing with fixed size packets. As seen, the operations used in this algorithm include basic addition, subtraction, division, and comparison instructions. However, the division always involves dividing by the flow's rate. But by specifying the rate as a fraction, and its inverse an integer, the division can be replaced by a multiplication.

5.2.2 Earliest Due Date

Under EDD, each packet is assigned a due date (deadline). Each flow i provides the minimum packet interarrival time I_i and a local delay bound d for each node the packet passes in the network. Assuming the k^{th} packet from flow i arrives at time $A(p_i^k)$, its deadline (or finish time) is $F(p_i^k) = A(p_i^k) + d$. In Delay EDD, an extension to EDD, the packet's dead-

```
(a) Variable length packets

$\sigma_{curr} = min\left\{\sigma_i, \sigma_{curr} + \left[\frac{A(p_i^k) - S(p_i^{k-1})}{\rho_i}\right]\right\}$
if $(L_i^k \leq \sigma_{curr})$ then
    $S(p_i^k) = A(p_i^k)$
    $\sigma_{curr} = \sigma_{curr} - L_i^k$
else
    $S(p_i^k) = A(p_i^k) + \frac{L_i^k - \sigma_{curr}}{\rho_i}$
```

```
(b) Fixed size packets

$\sigma_{curr} = min\left\{\sigma_i, \sigma_{curr} + \left[\frac{A(p_i^k) - S(p_i^{k-1})}{\rho_i}\right]\right\}$
if $(\sigma_{curr} > 0)$ then
    $S(p_i^k) = A(p_i^k)$
    $\sigma_{curr} = \sigma_{curr} - 1$
else
    $S(p_i^k) = S(p_i^{k-1}) + \frac{1}{\rho_i}$
```

Figure 5.1: Calculating eligibility times in a leaky-bucket shaper.

line is defined as $F(p_i^k) = max\{A(p_i^k) + d, F(p_i^{k-1}) + l_i\}$. In this algorithm only additions and comparison operations are required.

5.2.3 Fair Queueing

Packet-by-packet generalized processor sharing (PGPS), also known as Weighted FQ (WFQ), Frame-based FQ (FFQ), Starting-potential based FQ (SPFQ), Start-time FQ (SFQ), Worst-case Fair WFQ (WF2Q) and WF2Q+, and Self-clocked FQ (SCFQ) are several examples of packetized versions (PFQ) of the fair queueing (FQ) algorithm from the literature. Despite the large number of variations, the basic foundation for all these algorithms is the same. The function $V(t)$ returns the system time (or system potential) at time t. For each packet that enters the scheduler, it is assigned a start time $S(p_i^k)$ and a finish time $F(p_i^k)$. Packets are then transmitted in increasing order of start time or finish time, depending on the algorithm. The difference among the various PFQ algorithms is in how they update the system time, and how the start time and finish times are calculated.

For example, in PGPS

$$V(t+\tau) = V(t) + \frac{\tau}{\sum_{i \in B} \rho_i}, \text{ where B is the set of connections with queued packets,}$$

and

$$S(p_i^k) = max\{F(p_i^{k-1}), V(A(p_i^k))\}$$
$$F(p_i^k) = S(p_i^k) + \frac{L(p_i^k)}{\rho_i}$$

where $L(p_i^k)$ is the length of packet p_i^k.

In SCFQ the system time is defined as the finish time of the packet currently receiving service. SFQ uses the start time of the packet currently in service instead. In WF2Q+, $V(t+\tau) = max\{ V(t) + W(t,t+\tau), min_{i \in B}(S(p_i^{k(t)}))\}$, where $p_i^{k(t)}$ is the packet corresponding to the HOL packet for flow i at time t. The start times and finish times are computed in the same manner.

Similar to the leaky-bucket shaping algorithm, most PFQ algorithms require the same arithmetic instructions, along with a comparison operation. In WF2Q+, $min_{i \in B}(S(p_i^{k(t)}))$ is simply the top entry in the shaper PQ. This value is obtained by reading the highest priority entry (smallest eligible time) in the shaper PQ.

5.3 Basics

We assume that the buffer manager handles the actual buffering of the packet, and simply returns a packet tag which consists of the flow id, a pointer (starting address of the packet in the buffer, or page number for fixed size packet networks), and the packet length (in units of time slots, or not included for fixed size packet networks). These tags are placed into a FIFO queue, and are removed by the shaper-scheduler for processing.

The main memory elements are those for storage of flow state and flow linked list. First, the packet tag's flow id is used to access the flow's linked list. If this list is non-empty, then the tag is simply appended to the end of the list. Otherwise, this packet tag corresponds to a new HOL packet which requires a read of the flow's state to compute the packet's eligible time and finish time. Since the finish time needs to be computed anyway, doing so now instead of when the packet actually becomes eligible saves a memory read for the flow state. A new tag (shaper tag) consisting of the flow id, and its start and finish times are created and inserted into the shaper PQ. Depending on whether the one-stage or two-stage architecture is used, the tag is inserted into the scheduler PQ now or when the packet is transferred. A counter is also needed which is incremented at every time slot interval when data is transmitted, and stays idle otherwise.

As we saw in the previous section, all the service disciplines can be mapped to a common framework. The computation of the start and finish times requires one or more additions, comparison operations, and divisions (or multiplications). Since two operand min and max operations are common in these algorithms, separate min and max operations can be incorporated into the ALU. Instead of using an if-then-else statement, these instructions will reduce the number of cycles required to find the min and max of two operands. Finding eligible packets and link scheduling is performed by the shaper and scheduler PQs, so the controller simply needs to make a request to these modules. Based on this fact, a simple processor with an ALU consisting of a few instruction, several registers for temporary storage, and program memory is all that is required to implement any algorithm.

Figure 5.2 shows the required datapath. The data memory holds flow state, and its width depends on a cost/performance trade-off analysis. A wider memory unit will have better throughput, but will be more expensive to implement. Both memory units are written by an external management unit which will load in flow parameters and shaping and scheduling code. The packet transmission unit simply requires the starting address and length of the

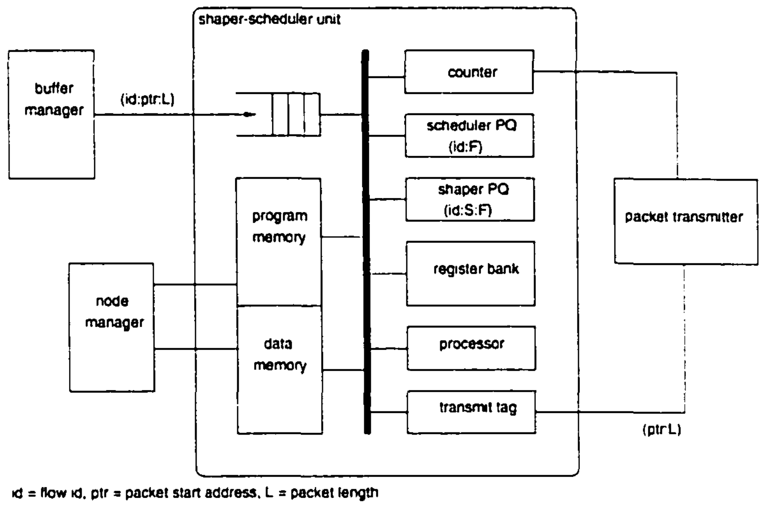


Figure 5.2: Logical block diagram of a generic shaper-scheduler implementation.

packet to transmit and sends back a signal which increments the counter after the transmission at each time slot.

5.4 Memory Management

Each algorithm requires different flow state information. For maximum throughput, all state information should be kept on a single line of memory addressed by the flow id. However, provisioning for the maximum possible amount of state information is unrealistic. This requires that multiple lines of memory be used, with the number of lines dependent on how much state is required. To address the flow state, a two stage memory access scheme is used. The first N lines of memory are reserved for pointers to the actual data, and are addressed by the flow id. The node manager will handle all memory management tasks. The flow's data is stored in contiguous lines starting at the address pointed by the returned pointer. This is shown in Figure 5.3. Within each line, one or more fields can be stored, and is determined by the node manager (e.g., a 64-bit wide memory can store two 32-bit fields). This arrangement requires that any range of the register be read/writable (i.e., a 64-bit register R0 is 4-way accessible if the values R0[15:0], R0[31-16], R0[47:32], R0[63:48] can be used as operands into the ALU). This method reduces memory requirements. The finish

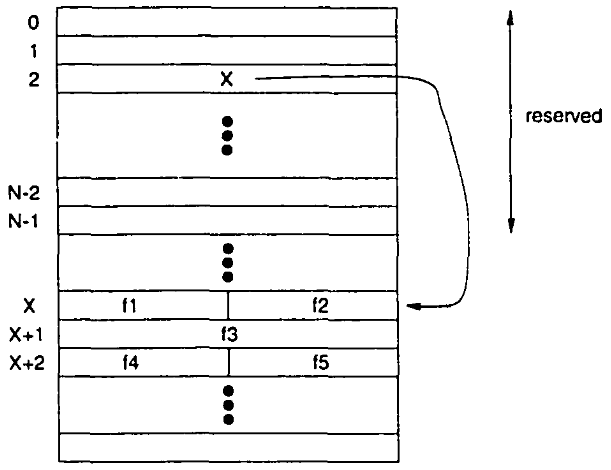


Figure 5.3: Memory management in data memory.

time might require a much larger number of bits than the token bucket size or packet length.

For simplicity we assume that the node manager will use the same shaping and scheduling discipline for all flows. This means that only one program needs to be stored in the program memory. This also simplifies the access of a particular field in the flow state, since the program needs to consider only one type of data structure for the flow's state. For example, as shown in Figure 5.3, to access f4 of flow i=2, the content of data memory at address 2 (DM[2]) is read, and the content of address (x+2) is read into a register. In this example, accessing the entire state of flow *i* will require (3+1) memory reads. Depending on the performance requirements, the width of the data memory can be increased to reduce the number of reads.

The data memory is also used to store the linked list of packet tags. The node manager reserves a portion of the data memory and allocates the necessary number of lines of memory (a page) for each packet tag. An idle address FIFO module stores the starting addresses of all pages which are not being used. This FIFO is initialized by the node manager. Upon entering, the new packet tag is stored in an empty page, whose address is first obtained from the idle address FIFO. When the packet corresponding to that packet tag has been sent to the packet transmitter, the page's address is returned to the idle address FIFO.

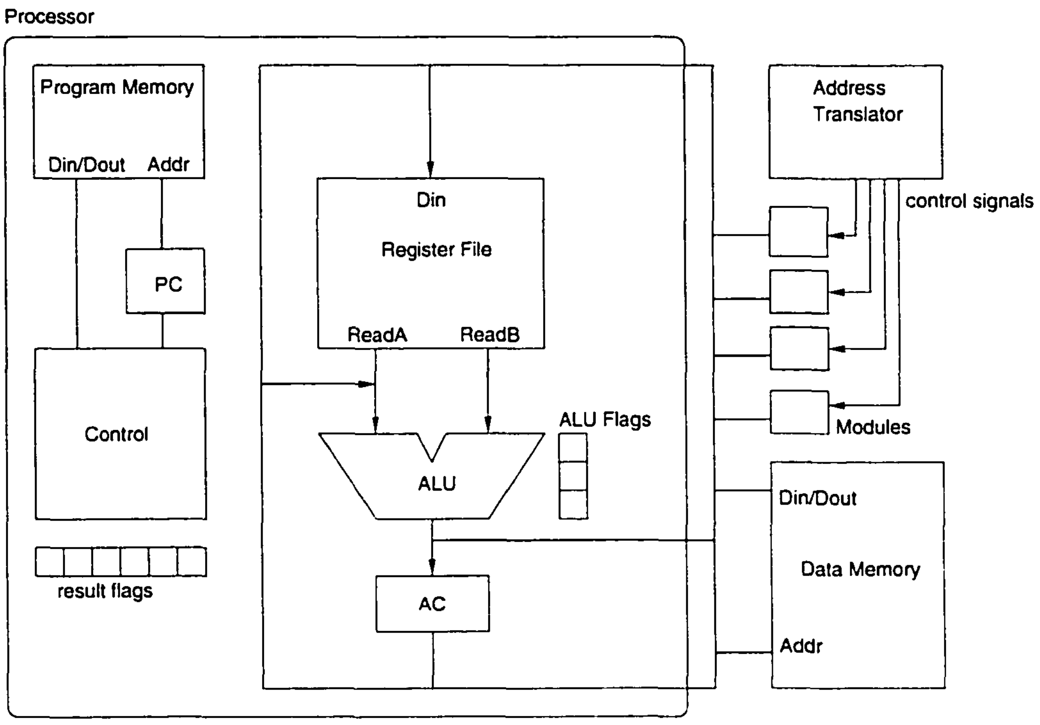


Figure 5.4: Block diagram of processing unit.

5.5 Processor

As mentioned previously, the processor needs to perform basic memory reads and writes, condition checking, comparisons (equal, greater, less than), branches, and arithmetic instructions. A max and min operation with two operands should also be included. Also, since the operands can be any field in the register, the ALU must be able to correctly select and operate on any range in the register. A memory mapped I/O scheme can be used to access the other modules in the datapath. A range of data memory can be reserved, with each module being mapped into one or more addresses in this range. A memory translator can be used to generate the correct control signals based on the address. For example, a read to address (N+1) can be mapped so that the shaper module will put the next eligible packet's tag onto the databus.

Figure 5.4 shows a block diagram of the processor. Here we assume a 64-bit wide 2-way accessible registers. The ALU will correctly handle operations with both 64 and 32 bit operands. For example, an add of a 32 bit and 64 bit operand will append the value 0 to the upper 32 bits of the 32 bit operand, with a 64 bit result. The bank of result flags are set by the module that was read/written by the processor. For example, a valid output from a shaper read will set the shaper_flag, and is cleared after it is checked by the processor. The counter_flag is set when the counter is incremented, and cleared when checked by the processor. The packet_flag is set when a packet is done transmitting, and cleared when checked by the processor.

5.6 Evaluation

The major cost of implementing the controller and datapath will be from the PQs and the memory units. Assuming the shaper-scheduler supports up to 1,000 flows, 64 bytes per flow state, 8 bytes per packet tag, and a shared packet buffer in the node for a maximum of 100,000 minimum sized packets, the total data memory required will be 872 KB for 64-bit wide memory (8 lines per flow state, 1 line per packet tag), and 468 KB for 32-bit wide memory (16 lines per flow state, 1 line per packet tag). For 10,000 flows these number increase to 1.52 MB and 1.08 MB. For 100,000 flows and a buffer space of 1 million packets, the memory requirements become 15.2 MB and 10.8 MB.

Since the number of instructions that need to be supported by the processor and the number of registers in the register file will be very small (less than 20 instructions and less than 10 registers are required), the cost of the processor will not be the deciding factor in overall implementation cost. However, this is not the case when we look at performance issues.

As we saw in previous sections, the shaper and scheduler PQ units are able to return results within 4 clock cycles. As we will see in Section 5.7, the processor can take more than 4 cycles to process each packet. However, we see that the attainable throughput by each shaper-scheduler is still more than adequate for today's high-speed networks. Assuming a clock period of 10 ns (100 MHz clock, which is very attainable with today's submicron CMOS technology), and assuming each packet processing takes on average 10 cycles (this also assumes the memory read/write latencies are within 10 ns), the shaper-scheduler implementation can process 10 million packets per second (4 Gbps throughput for 50-byte packets). Assuming an average packet length of 50 bytes, and in the worst-case scenario where 16 input ports are continuously forwarding packets, the maximum link speed that can be supported is 235 Mbps $((10\times10^6)\times 50\times 8/(16+1)$. If an algorithm requires slightly more processing of up to 20 cycles per packet, then this number drops to 118 Mbps. However, under more realistic conditions, and combined with a custom implementation of the shaper-scheduler using very aggressive design techniques and technology, we are confident that higher link speeds can be supported.

5.7 Example Implementation

This section presents implementation strategies for some well-known algorithms using our shaper-scheduler architecture. For each algorithm we present the program code assuming our two-stage shaper-scheduler PQ, and also the one-stage version. We will also discuss some implementation issues not covered in the previous section.

5.7.1 Preliminaries

The basic outlines for the two-stage and one-stage programs are shown in Figures 5.5 and 5.6. The main difference is that the two-stage version must transfer at most two tags from the shaper PQ to the scheduler PQ at every counter tick, while the one-stage version does not require this step. Also, both require a new scheduling decision after every packet transmission. Ideally, the packet flag is actually set before the packet is done transmitting, so that the next packet can be read from the packet buffer and immediately transmitted without any link idle time. At all other times both programs are processing newly-arrived packets.

5.7.2 Leaky-bucket shaper with static priority

A similar algorithm proposed in [88] shows the effectiveness of this type of shaping and scheduling in providing throughput, delay and delay jitter guarantees. The flow state consists of the max and current bucket size, rate, eligible time of the previous packet, and a

```
TOP: if counter_flag set {
   /*shaper to scheduler transfer*//*R0 holds system time*/
   M[SHAPER_V] = R0; /*write system time into shaper*/
   idle 3 cycles; /*wait for shaper*/
   R1 = M[SHAPER_R];
   if shaper_flag set {
      /*shaper output is valid*/
      M[SCHEDULER_W] = R1; /*write into scheduler*/
      /*check if there are more packets in this flow*/
      /*and update linked list*/
      /*compute start and finish times of new HOL packet*/
      /*and assemble tag in R1 and write tag into shaper*/
      M[SHAPER_W] = R1;
   }
   M[SHAPER_V] = R2; /*R2 holds S of current packet transmit*/
   idle 3 cycles;
   R1 = M[SHAPER_R];
   if shaper_flag set {
      /*shaper output is valid*/
      M[SCHEDULER_W] = R1; /*write into scheduler*/
      /*check if there are more packets in this flow*/
      /*and update linked list*/
      /*compute start and finish times of new HOL packet*/
      /*and assemble tag in R1 and write tag into shaper*/
      M[SHAPER_W] = R1;
   }
}
if packet_flag set {
   R1 = M[SCHEDULER_R];
   if scheduler_flag set {
      /*get packet length and ptr from linked list*/
      /*assemble a transmit tag in R1*/
      M[TRANSMIT_W] = R1;
   }
}
/*otherwise check FIFO for newly arrived packets*/
R1 = M[FIFO];
if fifo_flag set {
   /*if a HOL packet assemble tag in R3 and write into shaper*/
   /*otherwise update linked list*/
}
GOTO TOP;
```

Figure 5.5: Main program loop for the two-stage architecture.

```
TOP: if packet_flag set {
    M[SHAPER_V] = R0; /*write system time into shaper*/
                      /*R0 holds system time*/
    R1 = M[SCHEDULER_R];
    if scheduler_flag set {
       /*get packet length and ptr from linked list*/
       /*assemble a transmit tag in R1*/
       M[TRANSMIT_W] = R1;

       /*check if there are more packets in this flow*/
       /*and update linked list*/
       /*compute start and finish times of new HOL packet*/
       /*and assemble tag in R1 and write tag into shaper*/
       /*this will also cause a write into scheduler*/
       M[SHAPER_W] = R1;
    }
}
/*otherwise check FIFO for newly arrived packets*/
R1 = M[FIFO];
if fifo_flag set {
    /*if a HOL packet assemble tag in R3 and write into shaper*/
    /*otherwise update linked list*/
}
GOTO TOP;
```

Figure 5.6: Main program loop for the one-stage architecture.

static priority (which is the finish time in static priority schemes). The rate is given by two numbers *a* and *b* (*a* fixed size segments every *b* time slots). Since the inverse of the rate is needed, the data memory holds the value (*b/a*). This value will always be greater than zero, and will be rounded to the nearest integer. The flow state is shown in Figure 5.7.

Figure 5.8 shows the code which computes the eligible time based on the leaky-bucket shaping algorithm. The finish time in this case is a static priority, which can be read from the flow state.

5.7.3 Self-Clocked FQ

Figure 5.9 shows the code for a SCFQ scheduling algorithm, which only computes the finish time of the packet. As seen, only two line (13, 14) are required to compute the times-

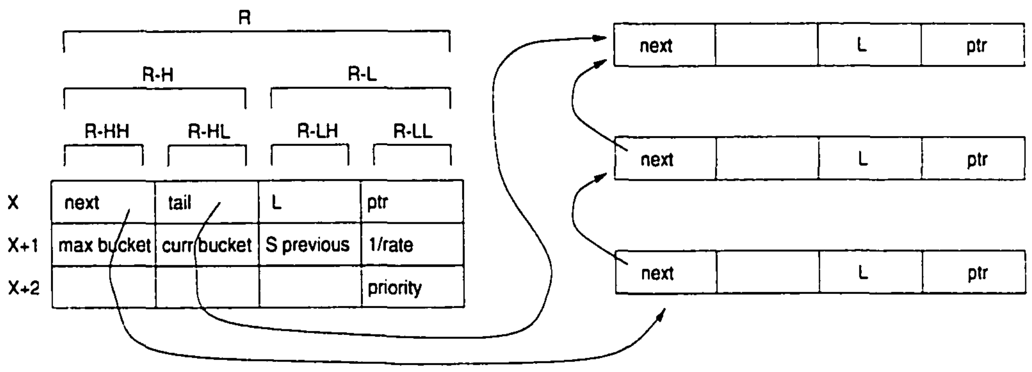


Figure 5.7: Flow state organization in data memory for a leaky-bucket with static priority service discipline.

```
/*R4-LL has the flow id. get addr to first line in R5*/
R5 = M[R4-LL];
R6 = M[R5]; /*get first line of state*/
R7 = M[R5 + 1]; /*2nd line*/
R8 = M[R5 + 2]; /*3rd line*/
AC = R0 - R7-LH;
AC = AC * R7-LL;
AC = AC + R7-HL;
R7-HL = min{R7-HH, AC}; /*curr bucket/
if R6-LH <= R7-HL { /*L<=current bucket*/
    R7-LH = R0; /*eligible time. R0 is system time*/
    R7-HL = R7-HL - R6-LH; /*updated bucket*/
}
else {
    AC = R6-LH - R7-HL;
    AC = AC * R7-LL;
    R7-LH = AC + R0; /*eligible time*/
}
R9-HL = R4-LL; /*create shaper tag:id*/
R9-LH = R7-LH; /*eligible time*/
R9-LL = R8-LL; /*priority*/
M[SHAPER_W] = R9; /*write into shaper*/
M[R5 + 1] = R7; /*update state*/
```

Figure 5.8: Computing eligible and finish time in leaky-bucket with static priority.

tamp. The remaining 5 lines are used to read and write the flow state. Depending on the

```
/*R0 has the F time of the packet currently transmit*/
/*   It is set when a scheduling decision is made*/
 HH       HL       LH                        LL
-------------------------------------------------------
| next   | tail   | L                        | ptr    | R6
--------------------------------------------------------
| unused | unused | finish of previous packet| 1/rate | R7
--------------------------------------------------------
/*R4-LL has the flow id. get addr to first line in R5*/
R5 = M[R4-LL];
R6 = M[R5]; /*get first line of state*/
R7 = M[R5 + 1]; /*2nd line*/
AC = max{R0, R7-LH};
AC = AC + (R6-LH * R7-LL); /*new finish time*/
R7-LL = AC;
M[R5 + 1] = R7; /*update state*/
```

Figure 5.9: Finish time computation in SCFQ.

performance requirements, the memory width can be increased, which can result in just 2 cycles for the memory accesses.

5.7.4 Weighted Round Robin (WRR)

In a WRR scheme, the amount of service each priority level receives during each round is based on its weight. Assuming each round has T service slots (each service slot is one flow), each flow is assigned a number *rank* (this determines the order in which flows are serviced in a given round), and a weight w_i (number of packets a flow is allowed to transmit per round). Along with these values, each flow also keeps track of the number of slots remaining in the current round. Based on this information, it is easy to compute the finish time of a packet, as shown in Figure 5.10. Since the scheduler transmits packets in increasing order of finish times, the scheduler will automatically jump over those flows with no packets to send. The finish time computation code is shown in Figure 5.11.

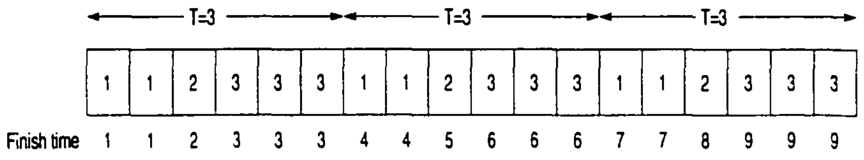


Figure 5.10: Finish time computation in weight round-robin. Flow 1 has weight 2, flow 2 has weight 1, and flow 3 has weight 3.

```
HH         HL              LH            LL
-----------------------------------------------
| next    | tail          | L           | ptr         | R6
-----------------------------------------------
| T       | slots remain | slots max | current F | R7
-----------------------------------------------
/*R4-LL has the flow id. get addr to first line in R5*/
R5 = M[R4-LL];
R7 = M[R5 + 1]; /*2nd line*/
R8-HL = R4-LL; /*R8 is new tag to be inserted in scheduler*/
R8-LH = R7-LL; /*R8 = id and finish time*/
R7-HL = R7-HL - 1; /*another slot used*/
if (R7-LH == 0) {
    R7-HL = R7-LH;
    R7-LL = R7-LL + R7-HH; /*next round*/
}
M[R5 + 1] = R7; /*update*/
```

Figure 5.11: Finish time computation in WRR.

5.8 Summary

In this chapter we examined and evaluated the design of a *complete* shaper-scheduler implementation which is flexible and scalable. Previous works has focused on just the sorting problem and examined mechanisms to determine eligible packets and move them into a scheduler queue. Whereas these efforts have simply targeted a small set of service disciplines, we propose a complete design which can efficiently support a wide range of algorithms. We first proposed two new architectures for queueing of packets and determining eligibility. For the first architecture, we proposed a new shaper PQ which can find the eligible packet with the smallest finish time in constant time. The second architecture eliminates the need for the shaper-to-scheduler transfer of eligible packets by using a unique one-stage solution. This is a deviation from the traditional, and more intuitive, two-stage solution that have been proposed. We evaluated these two new architectures at an architectural level and also discuss implementation issues and propose changes to the architecture to minimize some of the implementation related scaling effects (i.e., bus loading, multiple gate delays).We then describe a common theme upon which all shaping and scheduling algorithms can be mapped. Based on this analysis we propose a processor and datapath which can be programmed to implement any combination of shaping and scheduling algorithms. Using the results from previous research, and based on qualitative analysis, we show that our proposed architectures scale well to large number of flows, and with current CMOS technology, can be implemented to provide very high throughput for high-speed networks.

CHAPTER 6

HARDWARE/SOFTWARE CO-DESIGN AND CO-SIMULATION

6.1 Introduction

Integrating both hardware and software components during the modeling and simulation of a system allows a designer to easily evaluate various design choices and measure its impact on overall system performance. This chapter presents a simple event-based simulator which can handle both hardware and software components of a design without the need for expensive communication mechanisms, such as interprocess communications and exchange of control and data information, between the hardware and software components. Both the simulator and components are modelled in the C language, with software models annotated with timing-related information derived from processor-dependent specifications. The simulation of multiple software components is coordinated by a software component which models the task scheduling behavior of the target OS. Since all components are compiled into a single executable together with the simulator, simulation time is significantly reduced compared to simulators which interpret component descriptions during run time.

System designers typically need to partition a design into tasks performed by hardware components and those done in software running on a processor (e.g., general-purpose CPU, DPS chip). Traditionally, once the design has been partitioned, the hardware and software components are modelled and simulated separately using different modeling languages and simulation tools. Any design problems encountered later on during prototyping when both components are brought together becomes very costly to fix, and are usually patched up by adding extra code or logic. Also, the long turn-around time required to implement the repartitioning of the design makes it difficult to evaluate different partitioning strategies.

The problems mentioned above are well-known and have been the focus of numerous researchers tackling issues in hardware/software co-design and co-simulation. The work presented in this chapter describes a simulator which was developed while evaluating the impact of QoS (quality-of-service) support on network interfaces for streaming content servers. We needed to model a general-purpose server with a single network link connected through a network interface. QoS support could be implemented as dedicated hardware on the network interface, or as software running on an embedded processor on the network interface, or as software running on the server's CPU. While evaluating the impact of these design choices we found existing design tools inadequate or too slow. Our simulator allowed us to model and simulated arbitrarily large software components and arbitrarily small hardware components on a single platform. All components are modelled in C, while the simulator uses a single event-based queue with events sorted by time. Threads are used during the execution of software components, with a separate thread for the server's OS. Timing information is annotated into the original code to model delays in software components, while hardware components assume existence of a global clock. The C code for the software component is the exact same code (with very minor modifications) as the code that would be used in the actual implementation. After adding delay statements, the C code is compiled and executed on the local processor. Since both the simulator and all models are compiled into a single executable, run time is significantly reduced compared to other methods which interpret component descriptions during runtime. By combining the flexibility of C and the speed of a compiled simulation we were able to quickly explore different design alternatives and to obtain more meaningful data from running simulations of longer run times.

This chapter is organized as follows. We first look at some related work in hardware/software co-design and co-simulation in Section 6.2. We also describe different software performance estimation schemes in Section 6.2. In Section 6.3 we describe the basic operation of our event simulator and also illustrate details using the network interface as an example. Some performance results are shown in Section 6.4. Section 6.5 concludes the chapter with a summary of our work, and possible future extensions to the simulator.

6.2 Related Work

In developing our simulator we need to address two issues. In Section 6.2.1 we look at various schemes which provide a single environment for hardware/software co-design and co-simulation. In Section 6.2.2 we describe various methods to estimate software performance during hardware/software co-simulation.

6.2.1 Co-simulation

The natural choice to model hardware components is in a HDL (e.g., VHDL or Verilog), while high-level languages such as C or C++ are often used for software components. As such, one solution to developing a single environment for co-simulation simply combines both HDL simulations with C simulations. Examples of these include Seamless-CVE from Mentor Graphics and work presented in [50][79]. The separate simulations are synchronized via communication mechanisms, which further complicate the design process and adds overhead [85] during the simulation run. In [50] the HDL of hardware components are mapped to FPGAs. So instead of simulating the hardware components, the simulation actually runs the hardware models off FPGAs to speed up the simulation. Other solutions use separate clocks for the hardware and software simulations, with mechanisms to periodically synchronize the clocks.

A simple way to eliminate the need for synchronization is to use a single simulation. An obvious solution is to model the entire system, including the target processor which will run the software, using a single modelling tool. Execution of the software components running on the target processor is then simulated along with the hardware components. An example of this is SimOS [62], which models an entire system including the processor, memory hierarchy, and I/O devices to simulate the running of an entire OS and applications. One of the main drawbacks of this approach is that a new model needs to be written for each processor under consideration. Although the use of an ISS (instruction set simulator) reduces the design time, a significant amount of time is still required to model different processors. Also, the extra resources required to include the simulation of the processor model can significantly increase simulation time.

Alternative approaches to single simulators try to do without accurate models of the processors and ISSs by estimating delays of software components. Some related work in this area is detailed in Section 6.2.2. [72] models both hardware and software components in VHDL, which allows the simulation to run on any VHDL simulator. [53] uses the POLIS design environment [16] to describe and synthesize C code models of both hardware and software components. The Ptolemy tool [54] is then used to actually simulate the generated C code. Here, an event-based simulator sorts events based on timestamps and triggers the execution of the corresponding component. Global information keeps the status of shared resources (i.e., the processor running software components) to guarantee that only one software component is running on the processor at anytime. These methods provide an efficient means to quickly explore the different architectures for a given design. Although accuracy is sacrificed for simulation speed, researchers have shown that simulation results are relatively close to actual measured results. Since the main goal of these simulators is to quickly measure system performance at a high-level and early in the design stage, sacrificing some accuracy is an acceptable price to pay for significant reduction of simulation times.

Based on these results, we also use the single simulator with software performance estimation in our simulator. Although the simulator described in [53] is similar to ours, the mechanism used to simulate the software components is different. The simulator in [53] uses a *software scheduler* to determine which software component will execute next. This scheduler is not part of the design being simulated, and makes its decision based on the status of the processor running the software components. This implies that the event simulator needs to distinguish between hardware components and software components. In our simulator, we actually include a software component which models the target design's OS process scheduler. This software component handles the execution of all other software components running on a processor. As we will see in Section 6.3.1, this means the event simulator only needs to consider a generic component.

6.2.2 Software Performance Estimation

An accurate simulation of software requires a detailed model of the target processor. Because of the complexity of modern processors with widely-varying pipelines and memory hierarchies, such modeling is often impractical. An alternative is to generate the assembly code of the software component on the target processor. An ISS (instruction set simulator) is used during the simulation to interpret the behavior of each instruction and associate a delay with each instruction based on the target processor's profile [46]. Cycle-accurate ISSs also keep track of the state of the processor to give more accurate delays. State information can include the status of the pipeline, register usage, and cache and memory content. This method still requires a detailed knowledge of the target processor's instruction set. An alternative ISS [4] instead uses a virtual instruction set, whose delays are based on profiling target processors.

Although less accurate, annotating the software component with delay statements provides a much faster way to estimate delays incurred during execution. [43] compiles C code into assembly code, and then generates a C version of the assembly code (each C statement has a corresponding line in the assembly code) with delay statements associated with each assembly instruction. [33] does the same but also takes into account the number of memory reads and writes. [71] uses a flow graph of the software component and associates a delay for each node (which corresponds to a block of code). Execution of the software component can be mapped into a path along the graph, with the execution delay being simply the sum of the delays of the nodes along the path. While these approaches are dynamic in that delays are calculated during simulation, static methods [47] perform a static analysis of the software code before run time. However, such static analysis can only provide best case and worst case numbers.

We use a similar approach to that presented in [43]. However, instead of simulating the C version of the generated assembly code, we add the delay statements into the original C code, and directly compile and execute the code during simulation. [33] uses a thread per fixed code block (blocks of code with constant execution time regardless of inputs). A single software component could thus require multiple threads, with total execution time being the sum of the delays of all the threads that were run. Our simulator uses a single thread per software component, with delays accumulated after each C statement.

6.3 A Hardware/Software Co-Simulator

In this section we describe the simulator that we developed to study different network interface architectures with QoS support in the context of streaming multimedia servers. For this study we needed to simulate low-level hardware designs, software running on an embedded processor, and the server's OS and other programs running on the server processor. In order to obtain meaningful results we had to simulate long periods of time (about 15 minutes) instead of short times in the µsec or even msec range. Although Verilog was sufficient in modeling the hardware designs, it did not provide the constructs necessary to model software components. Also the time needed to run simulations was far too long to evaluate multiple configurations and work loads. By modeling all aspects of our design in C, and by compiling both the design and the simulator into a single executable, we were able to significantly reduce the time required for each simulation.

In Section 6.3.1 we describe our-event based simulator and the basic data structures. We also describe how we handle the simulation of software components. In Section 6.3.2 we use our network interface as an example to illustrate some of the details of our simulator.

6.3.1 Event-based Simulation

The basic mechanism of an event-based simulator is very simple. A queue maintains a list of events with each event corresponding to a component in the design. After processing an event, the simulator takes the event with the earliest time and removes it from the queue. The task associated with the corresponding component is executed, and any event that are triggered by this are inserted into the event queue which can be a simple linked list. More efficient techniques such as calendar queues can be used to speed up the insertion of new events into the queue. Since the number of components we had to model was small, consequently the number of events at any time was also small. For this reason, the linked list implementation was found to be sufficient for our simulator.

Figure 6.1 shows a basic view of the event queue. Each event consists of a timestamp, a pointer to a data structure (object) which corresponds to an instantiation of a component, and a pointer to the next event in the queue. An object consists of three parts. The first includes data structures which describe the current state of the component. These can correspond to the current state of a state machine, and memory elements such as FIFOs and registers. The second part contains a list of pointers to other objects which can be triggered

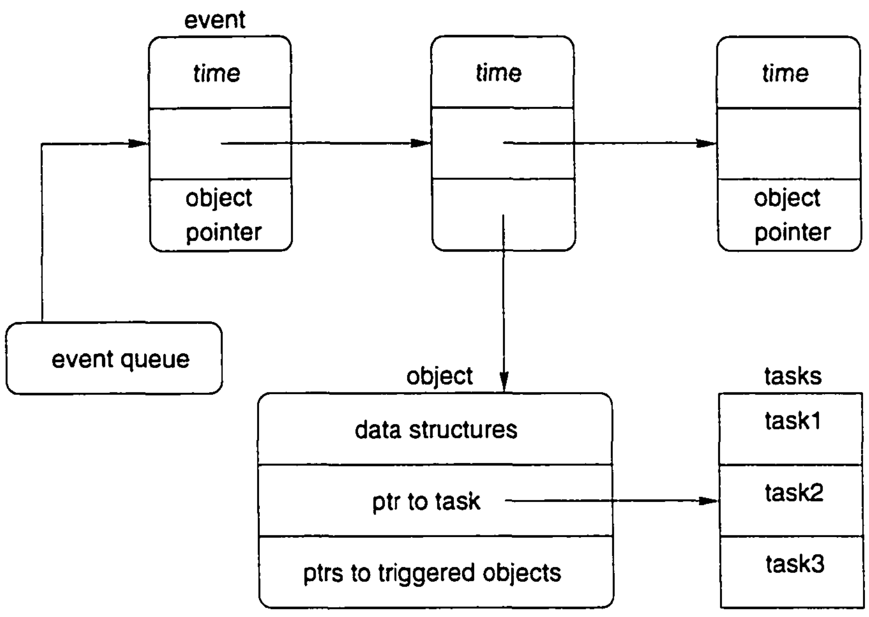


Figure 6.1: Basic simulator data structures.

by the execution of the current component. This simply corresponds to establishing a connection between component outputs to other component inputs. The last part is a simple pointer to the task which determines the behavior of the component. Multiple instantiations of the same object share the same code, so only a single copy of the code needs to be compiled into the simulator. When an event is removed from the queue, the simulator passes the pointer to the object as a parameter to the task and executes the task. The list of triggered object pointers are used to propagate any results to other connected components. At the end of the task, events corresponding to the triggered objects are created and returned to the event simulator.

Hardware components can be as simple as non-memoried combinational logic or complicated state machines with various memory and non-memory elements. In our design all our hardware components were of the latter. This substantially reduces the number of events and speeds up the simulation. The outcome of the execution of the task depends on the values in the data structures which describe the current state of the component. Since we assume all hardware components operate off of a global clock, passage of time is accumulated in clock cycles. The actual delay is obtained by multiplying by the clock period which is stored in a system configuration file. This allows us to easily change the clock

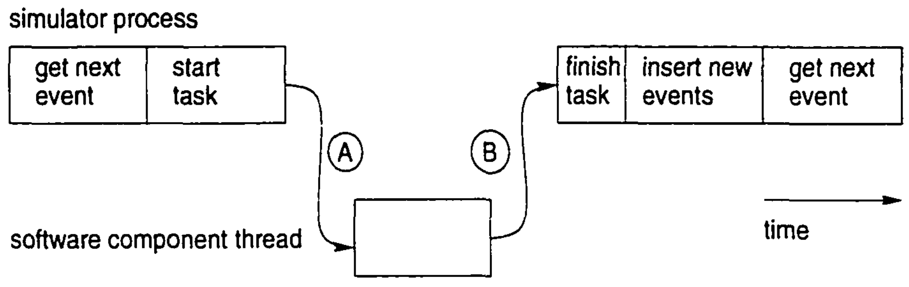


Figure 6.2: Simulation flow with a single software component thread.

speed for the entire system.

Unlike hardware components which operate concurrently with each other, execution of software components needs to take into account the status of the processor running the software component. The simulator needs to guarantee that only one process runs on a given processor at any time. In our simulator this is accomplished by the use of threads and semaphores. Each software component, including the server's OS, uses its own thread to execute its task. When a thread begins execution it locks the semaphore and hence blocks all other threads running on the same processor. This allows for easy facilitation of multiple processors in the simulation since threads can be assigned the semaphore corresponding to its processor. Figure 6.2 shows a simple example using a single software component. At the start of the simulation a thread for the software component is created and made to wait on its semaphore. At this point there are two processes in the simulation, one for the simulator and the other for the software component's thread, each with its own semaphore. The task for the software component wakes up the thread by performing a `sem_post` operation on the thread's semaphore and then blocks on a `sem_wait` call on its own semaphore (label A in Figure 6.2). The thread then runs for a while and returns control back to the simulator process by `sem_wait` on its semaphore and a `sem_post` on the simulator's semaphore (label B). At this point the thread returns any new events that it has triggered during its execution to the simulator. The delays used to determine the timestamp for these new events is dynamically calculated during the execution of the thread by accumulating clock cycles used for each C statement. After counting instructions from assembly code generated by gcc, we annotated the C code with delay statements. Although this method does not give

```
/* original C code               /* annotated C code
*/                               */

software_component(*object)      software_component(*object) {
{                                        delay(statement1);
        statement1;                      statement1;
        statement2;                      delay(statement2);
        statement3;                      statement2;
                                         break();
        ...                              delay(statement3);
                                         statement3;
}                                        break();

                                         ...

                                 }
```

Figure 6.3: Software component C code before and after annotations.

accurate measurements, we found the margin of error very reasonable and resulted in very fast simulation times. Also, accurate measurements in estimating the performance of the software components was not necessary since we were only interested in overall system performance early in the design process.

As described above, the software component's thread runs for sometime before returning control back to the simulator process. This amount of time can be very short (a few C statements) or long (several iterations of a loop) and depends on the amount of interaction between the software components and other components. During delay analysis of the C code, break-points are also added to the code, as shown in Figure 6.3. At each break-point a function call is made to block the thread (`sem_wait`) and wake up the simulator process (`sem_post`). Because we are using threads we do not need to worry about preserving variable contents or placing a marker to indicate where to resume execution. All of this is done by the threads, so when the thread is woken up it resumes execution from where it left off.

6.3.2 Example

Figure 6.4 shows a block diagram of our system which provides QoS functionality via software running on a processor which is on the network interface. Here we modelled a server

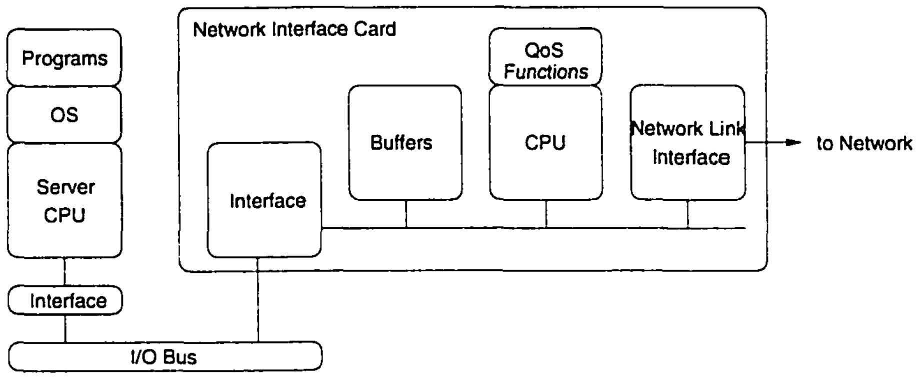


Figure 6.4: Block diagram of system with embedded processor on the network interface card.

process, an I/O interface (interface between the server processor and I/O bus, I/O bus and network interface, and communication mechanism used by the server process and network interface), a software component thread for the QoS functions (since this is the only task running on the embedded processor an OS and scheduler was not needed), and an interface to the network link.

At the start of the simulation the server process reads from a configuration file. This has a list of requests and times indicating when the request occurs. When the simulation time reaches a request's time, the server process downloads the necessary information into the network interface via the communication mechanism in the I/O interface. The QoS function, which runs continuously on the embedded processor, uses this information to schedule packets for transmission. The output of the QoS functions is a sequence of tags, which correspond to queued packets. These tags are written into the network link interface, which injects packets into the network.

An alternative design moves the QoS functions as a program running on the server processor. This program must compete for processor time with other programs. In our simulator we can create arbitrary number of load programs, which take up a certain amount of the processor time. This is done to measure the effect of different loads on the server processor. As seen in Figure 6.5 we have several threads, one for the OS's process scheduler, one for the QoS functions, and any number of load programs. Although there are several threads for each of the software components, only events for the process scheduler are needed in

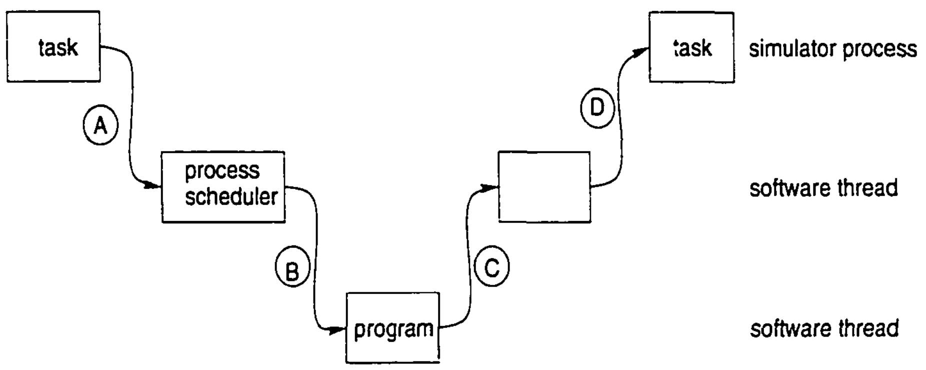


Figure 6.5: Simulation flow with multiple software components running on a single processor.

the event queue. The process scheduler keeps track of all other threads, which program was last to use the processor, and the amount of processor time each program has used so far. After the simulator wakes up the process scheduler (label A in Figure 6.5), a scheduling algorithm is used to determine which of the currently running programs will run next on the processor. The scheduling algorithm can be a simple round robin scheme, priority schemes, or an algorithm that takes into account the QoS requirements. The thread corresponding to this program is then awoken by the scheduler (B). After the program returns control to the scheduler (C), any events triggered by the program along with a new event for the scheduler are returned to the simulator (D). By having just process scheduler events in the event queue, events for each different program are not required. This results in a smaller number of events in the event queue and also as a consequence guarantees that only a single software component is simulated running on a given processor. We can easily add more processors to the simulation by adding more process schedulers.

Besides events for the components, we can also create events which do not correspond to any components in the design. For example, we can create debug events which are triggered periodically. These can be used to simply print the current simulation time, or to take snap-shots of the current state of components.

6.4 Performance

One of the main goals in developing a custom simulator was speed. Traditional tools that were available to us, such as Verilog and NCVerilog, were too slow. Some of the larger simulations took anywhere from 2-14 days to run on a Sun Ultra10 300 MHz workstation. This made it impossible to run the 30 configurations on each of the various implementation strategies. By modelling all components in C and compiling them into a single executable, we were able to achieve much faster simulation times. Pure hardware model simulations achieved the most improvement, with the longest simulation running for an hour. The addition of software components increased this time substantially to about 15-17 hours. This is due mainly to the high overhead of switching between threads.

As was mentioned earlier, our method of annotating simple delay statements into the software component code only produces rough estimates of the performance of the software. Accurate results require either modelling the processor in detail or keeping track of detailed status of the processor (e.g., pipeline stages, registers, cache, memory access latencies) during the execution of the software component. However, to measure the accuracy of our simulation model we performed tests on various different processors and compared the results to our estimated values. Figure 6.6 shows these results. The x-axis corresponds to different experiments. Each experiment uses different configuration parameters and system loads. The y-axis measures the amount of CPU time used for just the QoS functions. Values over 100% indicate that the processor was not fast enough, resulting in missed deadlines. As shown, the absolute values that our model estimated are not accurate. This was expected since we used a very crude algorithm in estimating software delays. What is important is that our model correctly estimates the trends. That is, our estimates correctly show a high-level view of CPU usage, which is adequate for evaluating different architectures early in the design stage with high-level descriptions. In fact, we can see that our estimates are very close to the values for the Pentium III 500. Such trend evaluation is important in evaluating the scaling properties of the software component and overall design early in the design process. For example, this allows the designer to quickly evaluate the load characteristics of a particular set of algorithms under consideration.

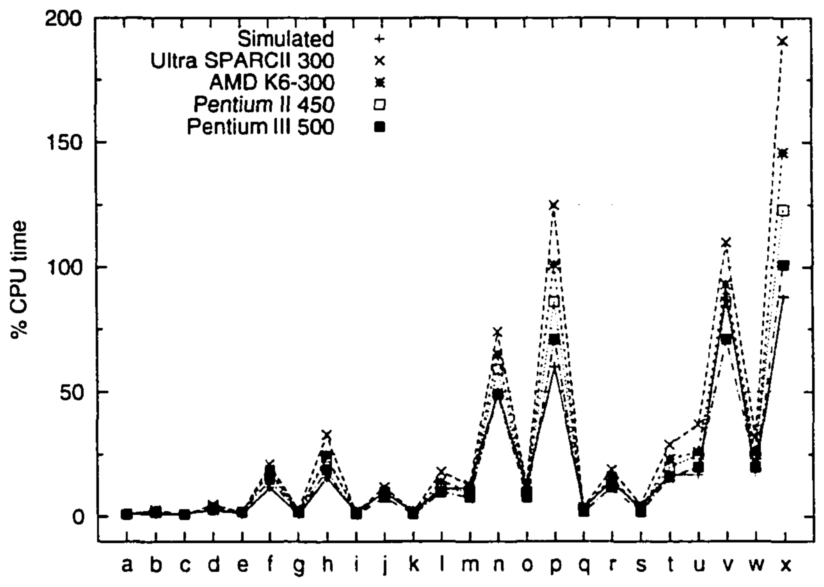


Figure 6.6: CPU usage comparison.

6.5 Summary

In this chapter we presented an event-based simulator for use in evaluating high-level designs with both hardware and software components. Simulation time is significantly reduced over other conventional simulation tools. This is accomplished by modelling all components in C and compiling these along with the simulator into a single executable. Software components are also compiled and run, instead of interpreted during the simulation. This provides a quick estimate of the software component's performance. Unlike other simulators, we model the actual implementation's OS process scheduler as a software component to keep track of the processor's state and to schedule the execution of the various software components on the processor. This feature has the benefit of allowing our simulator to easily support the simulation of multiple processors and multiple software components, and the switching of software components to different processors.

Although the current status of our simulator was sufficient for our needs, there are many more details that need to be worked out before it can become a general-purpose simulator. In this work we manually added delay and break statements in the software components.

Ideally, this would need to be automated, as it is a time-consuming process. We can also improve on the accuracy of the delay estimations by taking into consideration factors described in Section 6.2.2. We also need to deal with events that occur at the same time which affect the same objects. Besides these and other details, an important future work includes defining a clear set of structures from which all models will be derived. In other words, we need to define a set of guidelines for writing code for the hardware and software component descriptions. Although the current framework for writing design description code was sufficient for our designs, we still need to consider other designs to iron out details in our framework (interface between objects, events, and the simulator).

CHAPTER 7

CONCLUSIONS

7.1 Research Contributions

In this thesis, we proposed an architecture for integrating the traffic shaper and link scheduler. This thesis has made the following contributions made in developing this architecture.

Priority queue architectures: This forms the basic building block for this thesis. We presented two new basic architectures with constant-time operations for sorting. As we saw in later chapters, a high-performance priority queue is essential in any effective traffic shaping and link scheduling implementation. A detailed study explores both scalability and implementation issues related to these architectures.

Traffic shaper and link scheduler architectures: Using the work presented in Chapter 2 as a foundation, we show how extended versions of the priority queue architectures can be used in solving key issues in the design of a shaper-scheduler architecture. We extract the core mechanisms needed in all shaping and scheduling algorithms, and propose two new shaper-scheduler architectures. In particular, the second architecture incorporates a novel one-stage scheme which eliminates the need to transfer eligible packets from the shaper to the scheduler queue. We show how this architecture provides the same functionality as that of the traditionally accepted two-stage mechanisms.

Network interface architecture: We show how traffic shaping and link scheduling can be incorporated on an end-host server using dedicated hardware on the network interface. We present an implementation using an audio/video-on-demand streaming server application as an example. We show how the server can stream a large number of flows up to the maximum capacity of the server, while delivering low delay-jitter to the end-clients for uninterrupted playback of the streams.

A Shaper-scheduler processing (SSP) engine: We decompose traffic shaping and link scheduling algorithms to construct an instruction set which can be used to implement any mix of algorithms. A micro-controller-based processing engine is proposed that enables a network designer the maximal flexibility for selecting traffic shaping and link scheduling schemes. Incorporating our shaper-scheduler architectures from Chapter 3, we propose a *complete* architecture for implementing traffic shaping and link scheduling. This includes a flexible memory organization and datapath, and using our proposed architecture we show example implementations of several algorithms.

A hardware/software co-design and co-simulation tool: We examine the issues in designing and simulating a mixed design with both hardware and software components. Based an event-driven simulator core, our simulation tool uses threads to directly execute software components. Simulation time is further reduced by incorporating compiled simulation and software performance estimation techniques.

7.2 Future Directions

The increase in demand for network bandwidth continues to grow, even as network providers struggle to add capacity to existing infrastructure. This demand for more bandwidth is not expected to slow down anytime soon. More users connecting to the Internet through high-speed connections, introduction of new applications taking advantage of the Internet, increased use of high-bandwidth applications such as high-definition audio and video, third-generation wireless service which will connect the wireless world to the Internet, and new Internet appliances, are all expected to substantially increase network traffic. Today's routers are expected to handle aggregate throughputs of over 50 Gbps, with link speeds reaching 1 Gbps. To meet throughput demands, high-end routers employ a number of programmable hardware architectures to implement certain key router functions. Similar to our shaper-scheduler architecture, programmable IP route lookup engines are implemented in hardware to make quick route decisions at link speeds. However, as the diversity of applications using the Internet grows, routers will be required to meet the different QoS guarantees required by these applications. Our shaper-scheduler is just one piece of the entire solution.

Packet classification: Any QoS mechanism depends on quickly classifying a packet to determine the level of QoS it will receive. Similar to traffic shaping and link scheduling algorithms, the classification algorithm will need to change to meet different QoS goals. Flexible hardware architectures need to be developed to meet both performance and flexibility requirements. Also, it is not clear how different packet classification strategies affect delivered QoS to applications. We need to explore the performance of different packet classification strategies for a variety of traffic loads using a wide range of applications.

Router management: This central processing unit monitors and updates all control and data information on all router components. Along with route table generation and maintenance, it also needs to configure and update the various QoS mechanisms (e.g., our flexible shaper-scheduler). Effective router management software needs to be developed to take advantage of the underlying QoS mechanisms.

Traffic shaping and link scheduling performance: Our flexible architectures can be used to implement any mix of algorithms. However, we need to study the performance for various mixes of algorithms for a variety of traffic loads. Based on these performance results, new algorithms could be developed.

Router model: The issues mentioned above range from low-level hardware issues to high-level software issues. Using our simulator, a full model of a router can be used to quickly evaluate the efficacy of different mechanisms, evaluate trade-offs in hardware/software partitioning strategies, examine the effects of different router configurations, and determine new router features needed to deliver QoS.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] T. Abdelzaher, and K. G. Shin, "QoS Provisioning with qContracts in Web and Multimedia Server," *in Proceedings of Real-Time Systems Symposium*, pp. 44-52, 1999.

[2] C. M. Aras, J. F. Kurose, D. S. Reeves, and Henning Schulzrinne, "Real-Time Communication in Packet-Switched Networks," *Proceedings of IEEE*, vol. 82, no. 1, pp. 122-139, Jan 1994.

[3] C. Aurrecoechea, A. Campbell, and L. Hauw, "A Survey of QoS Architectures," *ACM/Springer Verlag Multimedia Systems Journal*, vol. 6, no. 3, pp. 138-151, May 1998.

[4] J. Bammi, E. Harcourt, W. Kruitzer, L. Lavagno, and M. Lazarescu, "Software performance estimation strategies in a system-level design tool," *in Proceedings of 18th Int'l Workshop on Hardware/Software Codesign*, pp. 82-86, 2000.

[5] D. Banks and M. Prudence, "High-performance network architecture for a PA-RISC workstation," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 191-202, February 1993.

[6] J. Bennett and H. Zhang, "Wf2q: Worst-case Fair Weighted Fair Queueing," *in Proceedings of IEEE INFOCOM*, pp. 120-128, 1996.

[7] J. Bennett and H. Zhang, "Hierarchical packet fair queueing algorithms," *in Proceedings of SIGCOMM*, pp. 143-156, 1996.

[8] R. Bhagwan and B. Lin, "Fast and Scalable Priority Queue Architecture for High-Speed Network Switches," *in Proceedings of IEEE INFOCOM*, pp. 538-547, 2000.

[9] M. Blumrich, C. Dubniki, E. Felten, and K. Li, "Protected, User-Level DMA for the SHRIMP Network Interface," *in Proceedings of Int'l Symp. on High-Performance Comp. Arch.*, pp. 154-165, 1996.

[10] N. Boden et al., "Myrinet: A Gigabit-per-Secong Local Area Network," *IEEE Micro*, vol. 15, no. 1, pp. 29-36, 1995.

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

[12] B. Carpenter and D. Kandlur, "Diversifying Internet delivery", *IEEE Spectrum*, vol. 36, no. 11, pp. 57-61, November 1999.

[13] J. Chao, "A Novel Architecture for Queue Management in the ATM Network," *IEEE Journal on Selected Areas in Communications*," vol. 9, no. 7, pp. 1110-1118, September 1991.

[14] H. J. Chao, Y. Jenq, X. Guo, and C. H. Lam, "Design of Packet-Fair Queuing Schedulers Using a RAM-Based Searching Engine," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1105-1126, June 1999.

[15] J. Chao, and N. Uzun, "A VLSI Sequencer Chip for ATM Traffic Shaper and Queue Management," *IEEE Journal of Solid-State Circuits*, vol. 27, no. 11, pp. 1634-1643, Nov 1992.

[16] M. Chiodo, P. Giusto, A. Jurecska, H. C. Hsieh, A. Sangiovanni-Vincentelli, and L. Lavagno. Hardware-software codesign of embedded systems. *IEEE Micro*, vol. 14, no. 4, pp. 26-36, August 1994.

[17] F. M. Chiussi and A. Francini, "Implementing Fair Queueing in ATM Switches: The Discrete-Rate Approach," *in Proceedings of INFOCOM*, pp. 272-281, 1998.

[18] F. M. Chiussi, A. Francini, and J. G. Kneuer, "Implementing fair queueing in ATM switches - Part 2: The logarithmic calendar queues," *in Proceedings of IEEE GLOBECOM*, pp. 519-525, 1997.

[19] D. D. Clark, S. Shenker, and L. Zhang, "Supporting Real-Time Applications in an Integrated Services Packet Network: Architecture and Mechanism," *in Proceedings of ACM SIGCOMM*, pp. 14-26, 1992.

[20] C. Dalton, G Watson, D. Banks, C. Calamvokis, A. Edwards, and J. Lumley, "Afterburner," *IEEE Network*, vol. 7, no. 4, pp. 36-43, July 1993.

[21] B. S. Davie, "The architecture and implementation of a high-speed host interface," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 228-239, February 1993.

[22] A. Demers, S. Keshav, and S. Shenker, "Analysis and simulation of a fair queueing algorithm," *in Proceedings of ACM SIGCOMM*, pp. 1-12, 1989.

[23] Z. Dittia, J. Cox, and G. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," *in Proceedings of IEEE INFOCOM*, pp. 179-187, 1995.

[24] P. Druschel, M. B. Abbott, M. Pragels, and L. L. Peterson, "Network subsystem design," *IEEE Network*, vol. 7, no. 4, pp. 8-17, July 1993.

[25] P. Druschel, and L. Peterson, "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility," *in Proceedings of 14th ACM Symposium on Operating System Principles*, pp. 189-202, 1993.

[26] P. Druschel, L. Peterson, and B. Davie, "Experience with a High-Speed Network Adaptor: A Software Perspective," i*n Proceedings of ACM SIGCOMM*, pp. 2-13, 1994

[27] D. Dunning et al., "The Virtual Interface Architecture," *IEEE Micro*, vol. 18, no. 2, pp. 66-76, 1998.

[28] W.-C. Feng, "Video-on-demand services: Efficient Transportation and Decompression of Variable Bit Rate Video," PhD thesis, The University of Michigan, 1996.

[29] W. Feng and J. Rexford, "A Comparison of Bandwidth Smoothing Techniques for the Transmission of Prerecorded Compressed Video," *in Proceedings of INFOCOM*, pp. 58-66, 1997.

[30] D. Ferrari and D. Verma, "A Scheme for Real-time Channel Establishment in Wide-area Networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 368-379, April 1990.

[31] S. J. Golestani, "A self-clocked fair queueing scheme for broadband applications," *in Proceedings of IEEE INFOCOM*, pp. 636-646, 1994.

[32] P. Goyal, H. Vin, and H. Cheng, "Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks," *IEEE/ACM Transactions on Networking*, vol. 5, no. 5, pp. 690-704, October 1997.

[33] R. K. Gupta and G. DeMicheli, "Constrained software generation for hardware-software systems," *in Proceedings of 3rd Int'l Workshop on Hardware/Software Codesign*. pp. 56-63, 1994.

[34] P. Gupta and S. Lin, "Routing Lookups in Hardware at Memory Access Speeds," *in Proceedings of IEEE INFOCOM*, pp. 1240-1247, 1998.

[35] P. Gupta and N. McKeown, "Packet classicification on multiple fields," *in Proceedings of SIGCOMM*, pp. 147-160, 1999.

[36] M. G. Hluchyj and M. J. Karol, “Queueing in High-Performance Packet Switching,” *IEEE Journal on Selected Areas in Communications*, vol. 6, no 9, pp. 1587-1597, Dec 1988.

[37] N. Huang and S. Zhao, "A Novel IP-Routing Lookup Scheme and Hardware Architecture for Multigigabit Switching Routers," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1093-1104, June 1999.

[38] C. Kalmanek, H. Kanakia, and S. Keshav, “Rate-Controlled Servers for Very High-Speed Networks,” *in Proceedings of GLOBECOM*, pp. 300.3.1-300.3.9, 1990.

[39] D. Kandlur, K. G. Shin, and D. Ferrari, “Real-time communication in multi-hop networks,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 10, pp. 1044-1056, October 1994.

[40] S. Keshav and R. Sharma, "Issues, and Trends in Router Design," *IEEE Communications Magazine*, pp. 144-151, vol. 36, no. 5, May 1998.

[41] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general-purpose ATM switch chip,” *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1265-1279, October 1991.

[42] V. P. Kumar and T. V. Lakshman, "Beyond Best Effort: Router Architectures for the Differentiated Services of Tomorrow’s Internet," *IEEE Communications Magazine*, pp. 152-164, vol. 36, no. 5, May 1998.

[43] M. Lajolo, M. Lazarescu, and A. Sangiovanni-Vincentelli, “A compilation-based software estimation scheme for hardware/software co-simulation,” *in Proceedings of 7th Int’l Workshop on Hardware/Software Codesign*, pp. 85-89, 1999.

[44] P. Lavoie, and Y. Savaria, “A Systolic Architecture for Fast Stack Sequential Decoders,” *IEEE Transactions on Communications*, vol. 42, no. 2/3/4, pp. 324-334, Feb/Mar/April 1994.

[45] C. E. Leiserson, “Systolic Priority Queues,” *Caltech Conference on VLSI*, pp. 200-214, 1979.

[46] J. Liu, M. Lajolo, and A. Sangiovanni-Vincentelli, “Software timing analysis using HW/SW cosimulation and instruction set simulator,” *in Proceedings of 6th Int’l Workshop on Hardware/Software Codesign*, pp. 65-69, 1998.

[47] S. Malik, M. Martonosi, and Y. S. Li, “Static timing analysis of embedded software,” *in Proceedings of 34rd annual conference on Design automation conference*, pp. 147-152, 1997.

[48] S.-W. Moon, J. Rexford, K. G. Shin, "Scalable hardware priority queue architectures for high-speed packet switches," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1215-1227, November, 2000.

[49] G. W. Neufeld, M. R. Ito, M. Goldberg, M. J. McCutcheon, and S. Ritchie, "Parallel host interface for an ATM network," *IEEE Network*, pp. 24-34, July 1993.

[50] K. A. Olukotun, R. Helaihel, J. Levitt, and R. Ramirez, "A software-hardware cosynthesis approach to digital system simulation," *IEEE Micro*, vol .14, no. 4, pp. 48-58, 1994.

[51] S. Pakin, V. Karamcheti, and A. Chien, "Fast Messages: Efficient, Portable Communication for Workstation Clusters and MPPs," *IEEE Concurrency*, vol. 5, no. 2, pp. 60-73, 1997.

[52] A. K. J. Parekh, "A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks", PhD thesis, Massachusetts Institute of Technology, 1992.

[53] C. Passerone, L. Lavagno, M. Chiodo, and A. Sangiovanni-Vincentelli, "Fast Hardware/software Co-simulation For Virtual Prototyping And Trade-off Analysis," *in Proceedings of 34th Design Automation Conference*, pp. 389-394, 1997.

[54] C. Passerone, L. Lavagno, C. Sansoe, M. Chiodo, and A. Sangiovanni-Vincentelli, "Trade-off Evaluation in Embedded System Design via Co-simulation," *in Proceedings of ASP-DAC*, pp. 291-297, 1997.

[55] T. Pei and . Charles. Zukowski, "Putting Routing Tables in Silicon," *IEEE Network Magazine*, pp. 42-50, vol. 6, no. 1, January 1992.

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

[57] K. K. Ramakrishnan, "Performance consideration in designing network interfaces," *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 203-219, February 1993.

[58] J. Rexford, F. Bonomi, A. Greenberg, and A. Wong, "Scalable architectures for integrated traffic shaping and link scheduling in high-speed ATM switches," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 5, pp. 938-950, June 1997.

[59] J. Rexford, J. Hall, and K. G. Shin, "A Router Architecture for Real-Time Point-to-Point Networks," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1088-1101, October 1998.

[60] J. Rexford, A. G. Greenberg, and F. G. Bonomi, "Hardware-Efficient Fair Queueing Architectures for High-Speed Networks," *Proceedings of IEEE INFOCOM*, pp. 638-646, 1996.

[61] J. Rexford, S. Sen, J. Dey, W. Feng, J. Kurose, J. Stankovic, and D. Towsley, "Online smoothing of live, variable-bit-rate video," *in Proceedings of International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pp. 249-257, 1997.

[62] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete Computer Simulation: The SimOS Approach," *IEEE Parallel and Distributed Technology*, vol. 3, no. 4, pp. 34-43, 1995.

[63] D. Saha, "Supporting distributed multimedia applications on ATM networks," PhD thesis, The University of Maryland, 1995.

[64] J. D. Salehi, Z.-L. Zhang, J. F. Kurose, and D. Towsley, "Supporting stored video: Reducing rate variability and end-to-end resource requirements through optimal smoothing," *Performance Evaluation Review*, vol. 24, no. 1, pp. 222-231, May 1996.

[65] P. Steenkiste, "A Systematic Approach to Host Interface Design for High-Speed Networks," *IEEE Computer*, pp. 47-57, March 1994.

[66] D. C. Stephens, J. Bennett, and H. Zhang, "Implementing Scheduling Algorithms in High-Speed Networks," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1145-1158, June 1999.

[67] D. Stiliadis, "Traffic Scheduling in Packet Switched Networks: Analysis, Design, and Implementation," PhD thesis, University of California at Santa Cruz, 1996.

[68] D. Stiliadis and A. Varma, "Efficient fair queueing algorithms for packet-switched networks," *IEEE/ACM Transactions on Networking*, vol. 6, no. 2, pp. 175-185, April 1998.

[69] D. Stiliadis and A. Varma, "Frame-based Fair Queueing: A New Traffic Scheduling Algorithm for Packet-Switched Networks," *in Proceedings of SIGMETRICS*, pp. 104-115, 1996.

[70] D. Stiliadis and A. Varma, "A general methodology for designing efficient traffic scheduling and shaping algorithms," *in Proceedings of INFOCOM*, pp. 326-335, 1997.

[71] K. Suzuki and A. Sangiovanni-Vincentelli, "Efficient software performance estimation methods for hardware/software codesign," *in Proceedings of 33rd Design Automation Conference*, pp. 605-610, 1996.

[72] B. Tabbara, M. Sgroi, A. Sangiovanni-Vincentelli, E. Filippi, and L. Lavagno, "Fast hardware-software co-simulation using VHDL models," *in Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, pp. 309-316, 1999.

[73] F. A. Tobagi, "Fast Packet Switch Architectures for Broadband Integrated Services Digital Network," *Proceedings of the IEEE*, vol. 78, no. 1, pp 133-167, Jan 1990.

[74] K. Toda, K. Nishida, E. Takahashi, N. Michell, and Y. Yamaguchi, "Design and Implementation of a Priority Forwarding Router Chip for Real-Time Interconnection Networks," *International Journal of Mini and Microcomputers*, vol. 17, no. 1, pp 42-51, 1995.

[75] D. Towsley, "Providing Quality of Service in Packet Switched Networks," *Performance Evaluation of Computer and Communication Systems*, L. Donatiello and R. Nelson, editors, pp. 560-586, Springer Verlag, 1993.

[76] L. Trajkovic and S. J. Golestani, "Congestion control for multimedia services," *IEEE Network*, vol. 6, no. 5, pp. 20-26, September 1992.

[77] C. B. S. Traw, and J. M. Smith, "Hardware/software organization of a high-performance ATM host interface", *IEEE Journal on Selected Areas in Communications*, vol. 11, no. 2, pp. 240-253, February 1993.

[78] J. Turner, "New directions in communications, or which way to the information age?", *IEEE Communication Magazine*, vol. 24, no. 19, pp. 8-15, October 1986.

[79] C. Valderrama, A. Changuel, and A. Jerraya, "Virtual prototyping for modular and flexible hardware/software systems," *Journal of Design Automation for Embedded Systems*, Kluwer Academic Publishers, vol. 2, no. 3/4, pp. 267-282, 1997.

[80] A. Varma and D. Stiliadis, "Hardware implementation of fair queuing algorithms for asynchronous transfer mode networks," *IEEE Communications Magazine*, vol. 35, no. 12, pp. 54-68, December 1997.

[81] D. Verma, H. Zhang, and D. Ferrari, "Delay jitter control for real-time communication in a packet switching network," *in Proceedings of IEEE TRICOMM*, pp. 35-43, 1991.

[82] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-Net: A User-Level Network Interface for Parallel and Distributed Computing," *in Proceedings of ACM Symposium on Operating Systems Principles*, pp. 40-53, 1995.

[83] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable High Speed IP Routing Lookups," *in Proceedings of ACM SIGCOMM*, pp. 25-36, 1997.

[84] T. Woo, "A Modular Approach to Packet Classification: Algorithms and Results," *in Proceedings of IEEE INFOCOM*, pp. 1213-1222, 2000.

[85] S. Yoo and K. Choi, "Synchronization Overhead Reduction in Timed Cosimulation," *in Proceedings of IEEE International High Level Design Validation and Test Workshop*, 1997.

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

[87] H. Zhang and D. Ferrari, "Rate-Controlled Service Disciplines," *Journal of High Speed Networks*, vol. 3, no. 4, pp. 389-412, 1994.

[88] H. Zhang and D. Ferrari, "Rate-Controlled Static-Priority Queueing," *in Proceedings of IEEE INFOCOM*, pp. 227-236, 1993.

[89] H. Zhang and S. Keshav, "Comparison of Rate-Based Service Disciplines," *in Proceedings of ACM SIGCOMM*, pp. 113-121, 1991.

[90] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet-Switched Networks", *ACM Transactions on Computer Systems*, vol. 9, no. 2, pp. 101-124, May, 1991.

[91] L. Zhang, B. Beacham, M. Hashemi, P. Chow, and A. Leon-Garcia, "A Scheduler ASIC for a programmable packet switch," *IEEE Micro*, vol. 20, no. 1, pp. 42-48, January 2000.

ABSTRACT

HARDWARE SUPPORT FOR
QUALITY-OF-SERVICE GUARANTEES
IN PACKET SWITCHED NETWORKS

by
Sung-Whan Moon

Chair: Kang G. Shin

Modern integrated networks can support the diverse quality-of-service requirements of current and emerging applications by incorporating effective traffic shaping and link scheduling mechanisms. However, processing a large number of packets on a high-speed link requires an efficient hardware implementation of the shaping and scheduling mechanism. This thesis presents new hardware architectures which provide fast, flexible, and efficient implementations for delivering QoS guarantees.

Priority queues are essential in all traffic shaping and link scheduling algorithms, and their efficacy is dependent on an effective priority queue mechanism.We propose two new priority queue architectures that scale to the large number of packets and large number of priority levels necessary in modern switch designs.

Building on these results, we propose two new traffic shaper and link scheduler architectures. By incorporating our programmable shaping and scheduling processor, we propose a *complete* traffic shaper and link scheduler implementation which achieves high performance and maximum flexibility needed to implement a wide range and mix of shaping and scheduling algorithms. We also investigate traffic shaping and link scheduling issues on an end-host server, and propose a network interface architecture with dedicated shaping and scheduling support.

Finally, we describe a hardware-software codesign and co-simulation tool which we developed to implement our architectures. The tool allowed us to evaluate both low-level hardware and high-level software components of a design using a common platform.

Incorporating event-driven simulation, software performance estimation, and compiled simulation techniques, we were able to easily evaluate different software/hardware partitioning strategies.