Concurrent Implementation of Packet Processing Algorithms on Network Processors

by

Mark Groves

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2006

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Mark Groves

Abstract

Network Processor Units (NPUs) are a compromise between software-based and hardwired packet processing solutions. While slower than hardwired solutions, NPUs have the flexibility of software-based solutions, allowing them to adapt faster to changes in network protocols.

Network processors have multiple processing engines so that multiple packets can be processed simultaneously within the NPU. In addition, each of these processing engines is multi-threaded, with special hardware support built in to alleviate some of the cost of concurrency. This hardware design allows the NPU to handle multiple packets concurrently, so that while one thread is waiting for a memory access to complete, another thread can be processing a different packet. By handling several packets simultaneously, an NPU can achieve similar processing power as traditional packet processing hardware, but with greater flexibility.

The flexibility of network processors is also one of the disadvantages associated with them. Programming a network processor requires an in-depth understanding of the hardware as well as a solid foundation in concurrent design and programming. This thesis explores the challenges of programming a network processor, the Intel IXP2400, using a single-threaded packet scheduling algorithm, SI-WF^{2}Q, as a sample case. SI-WF^{2}Q is a GPS approximation scheduler with constant time execution. The thesis examines the process of implementing the algorithm in a multi-threaded environment, and discusses the scalability and load-balancing aspects of such an algorithm. In addition, optimizations are made to the scheduler implementation to improve the potential concurrency. The synchronization primitives available on the network processor are also examined, as they play a significant part in minimizing the overhead required to synchronize memory accesses by the algorithm.

Acknowledgments

First and foremost, I would like to thank my supervisor Martin Karsten. His guidance kept me on track through numerous setbacks and his insight into the problems I encountered was invaluable in finding solutions. I would also like to thank Peter Buhr for reviewing my early drafts and providing feedback that made this thesis more comprehensible, and my thesis readers for their comments on my completed work. Finally, I would like to thank my family and Alex for their love and support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are two main solutions for processing packets in a networked environment. First, using a general-purpose processor, like the processor that powers a desktop computer. This solution has the advantage that it is cheap, has a large software base, can deal with virtually any network protocols and can be easily upgraded with new protocols. These advantages mean that software-based packet processing products can be put out on the market quickly, and easily upgraded with additional functionality.

However, the general processor solution is unable to keep up with improvements in network speeds due to the comparatively slow improvements in the processor and the data bus that connects it to memory. This problem is due to what is termed the *memory wall* [51], alluding to the fact that data-bus speeds and corresponding memory speeds do not increase at the same rate as processor speeds. With each upgrade in processor speed, this means that more processor cycles are spent waiting for the results of memory accesses to complete and return to the processor via the memory bus. Since all network data must flow through the bus to the CPU, the bus becomes the limiting factor in performance, without some direct connection between the network controller and CPU.

Packet processing tasks have also become more complex. Initially, routers just had to decrement the Time-To-Live field, perform a route lookup and recompute a checksum. These simplistic tasks could be easily handled, albeit slowly, using a general-purpose processor solution. With new protocols and methods of communication, some routers now have to handle encrypted packets, Quality of Service (QoS), as well as evaluate congestion and perform other computationally intensive operations. While general processors are designed to be more multi-purpose, they cannot handle the speed required by fast data networks.

The inability of general processors to handle packets at high rates leads to the second, and now preferred, solution for high-speed packet processing: Application Specific Integrated Circuits (ASICs). These customized ASICs are able to handle packets at wire-speed, which is the most important requirement of packet processing solutions. Unfortunately, the high-speed data buses

necessary to achieve line speed make the ASIC a very expensive solution, even when produced in high volume. Also, ASICs are slow to design and develop, and thus have a longer time-to-market, making them cost-inefficient with respect to production and manufacturing. Given the rate of speed increases in a network, each ASIC platform has a short lifetime in the market. Finally, since these are hardwired platforms, they are inflexible, making them unable to handle changes to protocols or have additional functionality attached after manufacturing. Nevertheless, ASICs represent the only solution that is able to handle packets at high speed and so the Internet has come to rely on these devices.

Following the increase in network speeds, new applications and services have been developed to utilize the newly available bandwidth. Voice packets travel over IP networks, packets can be tagged with additional data to help classify them for different services and proxies monitor web site requests to reduce the amount of traffic at the site. These real-time applications have minimum requirements that the original best-effort nature of the Internet was not designed for, and so this functionality must now become part of the functionality in Internet routers. Developing hardwired solutions for all these tasks is possible, but with each new service or extension of existing service, a new piece of hardware needs to be designed, tested and manufactured. In contrast, software based solutions on a typical PC machine can be programmed to execute any of these tasks, but again, not at the high transmission speed required by today's networks.

The introduction of Network Processor Units (NPUs) in the late 1990s creates a third solution for packet processing products. Network processors are a hybrid between the two previous solutions, making it possible to develop software based solutions as on a PC-based packet processor, while still using specialized hardware capable of handling packet processing tasks at the current wire-speed of networks. Additionally, because all processing is software based, there is no need to develop new expensive hardware (except for significant changes in network design), which keeps the cost and time-to-market small. Finally, since the same hardware can be used for many diverse applications, these network products will hopefully become standard commodities, produced in high volume, which should drive costs even lower.

However, the biggest challenge in using NPUs is programming them. Development environments are still in their infancy, as are the programming languages for these devices. Designing processing applications is difficult because the application must be divided into discrete tasks executed in limited bursts. Moreover, managing the multiple forms of parallelism is very difficult for all but the most basic processing tasks, but this parallelism is necessary to achieve high levels of performance. For example, there is often a complex interaction of tasks and memory accesses that must be balanced to be able to reach the maximum processing speed without overloading any of the components. In these cases, even a very simple and straightforward task can become extremely complex when attempting to achieve maximum performance.

1.1 Motivation

This thesis examines the complexity of programming NPUs through the development of a novel scheduling algorithm [28] on the Intel IXP2400 Network Processor [7], and the adjustments and optimizations necessary to fully utilize the hardware. The algorithm uses sets of timer wheels in a pipeline to schedule packets in constant time, which is shown to translate quite well to the IXP2400 architecture. The two main steps of this thesis are translating the sequential packet scheduling algorithm to a parallel algorithm that uses the multiple threads in the IXP2400, and the implementation of the multi-threaded algorithm in Intel's network processor assembly programming language. In particular, this thesis details the concurrency and synchronization requirements of porting the scheduler to a multi-threaded algorithm. Caching and other optimizations are also examined in order to develop an understanding of the tradeoffs involved with managing shared data and processing tasks.

1.2 Outline

This thesis is structured as follows: Chapter 2 give a brief background on network processors, especially the IXP2400. Chapter 3 provides a brief overview of related technology and the work being done with network processors. Chapter 4 presents an overview of the concurrency mechanisms available on the IXP2xxx series of network processors and Chapter 5 discusses possible adaptations of the original scheduler design and details the process of transforming the algorithm to a multi-threaded one and implementing it. Chapter 6 details the specific implementation of one adaptation, as well as some bottlenecks of the scheduler and what optimizations are possible to improve the packet processing rates. Finally, Chapter 7 looks at the conclusions of this work and details future plans and further improvements that could be made to the system.

Chapter 2

Network Processors

2.1 General Architecture

A Network Processing Unit (NPU) is a hybrid between a general purpose processor and a hard-wired architecture. Their goal is to achieve some level of general programmability while simultaneously achieving similar performance to ASIC-based packet processors. To reach this goal, compromises are made between the two objectives. For example, NPUs have a special instruction set to assist with common packet processing tasks, such as striping a packet of its headers, classifying them or performing route lookups. In addition to a special instruction set, NPUs also have specialized hardware blocks to aid in packet processing, but that are kept separate from the processors in order to maintain generality. Finally, rather than the customized memory and data buses in a hardware router, network processors use a combination of commodity and specialized memory to speed up memory accesses.

2.1.1 General NPU Architectures

As of yet, no general consensus has been reached on what hardware is necessary to achieve optimal packet processing, but there are some trends. Currently, network processor manufacturers attempt to distinguish themselves within the market by the feature set of their network processors, which unfortunately results in only a few commonalities in their architectures. All NPUs have special hardware blocks to speed up common network processing tasks, but the tasks are usually different. These hardware blocks can be used to perform complex cryptographical calculations, calculate checksums or generate timestamps. Since handling traffic regularly involves looking up the final destination of a packet and determining what the next hop is, some network processors have special hardware tables to aid in the lookup process. Also, to schedule and shape the traffic, there are often hardware assisted memory queues for storing packets. With so many choices and the tradeoffs among different NPUs, the needs of the user largely define the best network processor to use in any given solution. However, the choice of hardware can also correspondingly reduce

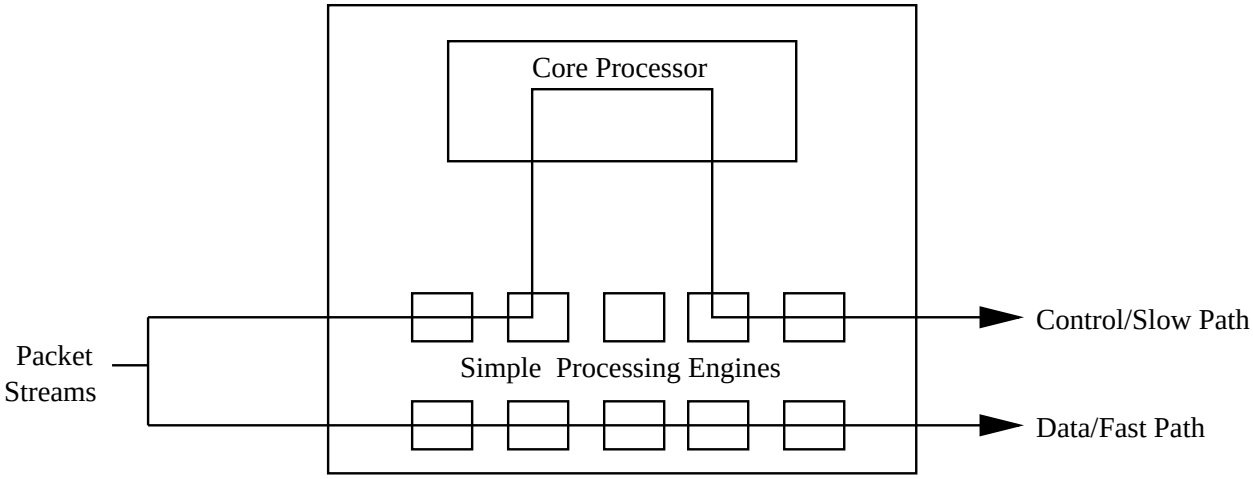


Figure 2.1: A General NPU Architecture with Data and Control Paths

the flexibility available to the user, since the lack of general consensus regarding architectures can lock the user into a specific brand of network processor.

The common trends in network processor architectures are all due to the similar obstacles that must be overcome. First of all, the constant increase in network speeds, faster than increases in processor speeds, forces designers to use any inherent concurrency available when processing packets. During packet processing, each packet generally gets the same treatment as it travels through a processor. In addition, this treatment, while complicated, is still relatively straight-forward. This straight-forward packet processing is generally called the fast or data path and is the path that the vast majority of packets travel on the way to their final destination. To handle the high volume of traffic and the relatively simple requirements of processing it, NPU designers often use multiple simple processing engines to handle this traffic. Possessing a small storage space for instructions, these engines can perform the basic processing for the fast path, and perform it on multiple packets simultaneously. Besides the data path, there is also the control path. The control path handles packets that are destined for the packet processor itself, such as Border Gateway Protocol (BGP) routing updates. These packets are handled using more complex algorithms than the minimal fast-path processors can provide. These control packets are often routed to a more general processor, which is either part of the NPU or on a separate card. Figure 2.1 illustrates how the data and control paths are handled within an NPU.

Another common obstacle is that any memory access takes a certain number of cycles to look up and return a result via a separate bus, time during which the processor is effectively idle. It is the need to minimize the number of cycles taken up by memory accesses that necessitates the specialized, expensive hardware in hard-wired routers. Many network processors attempt to

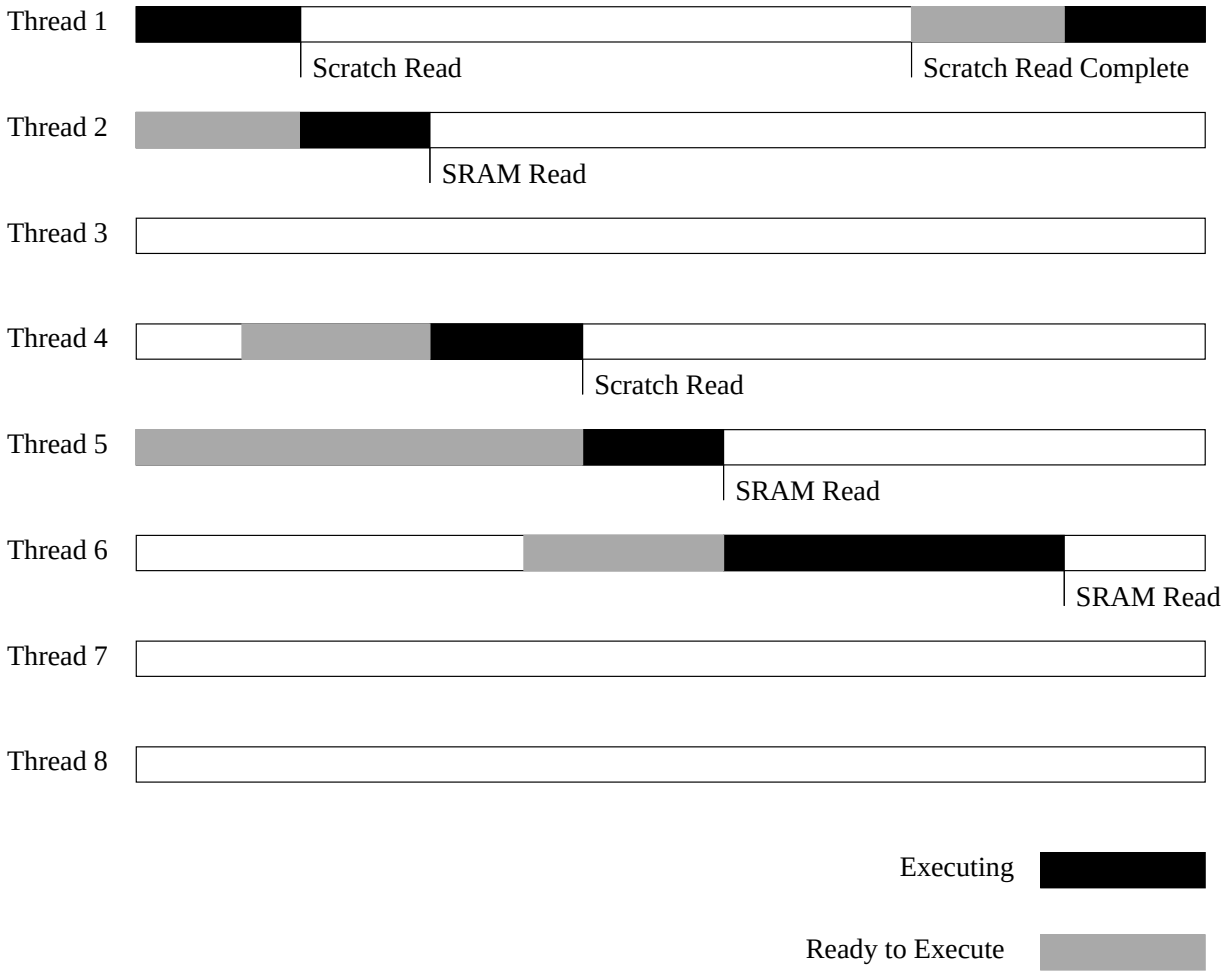


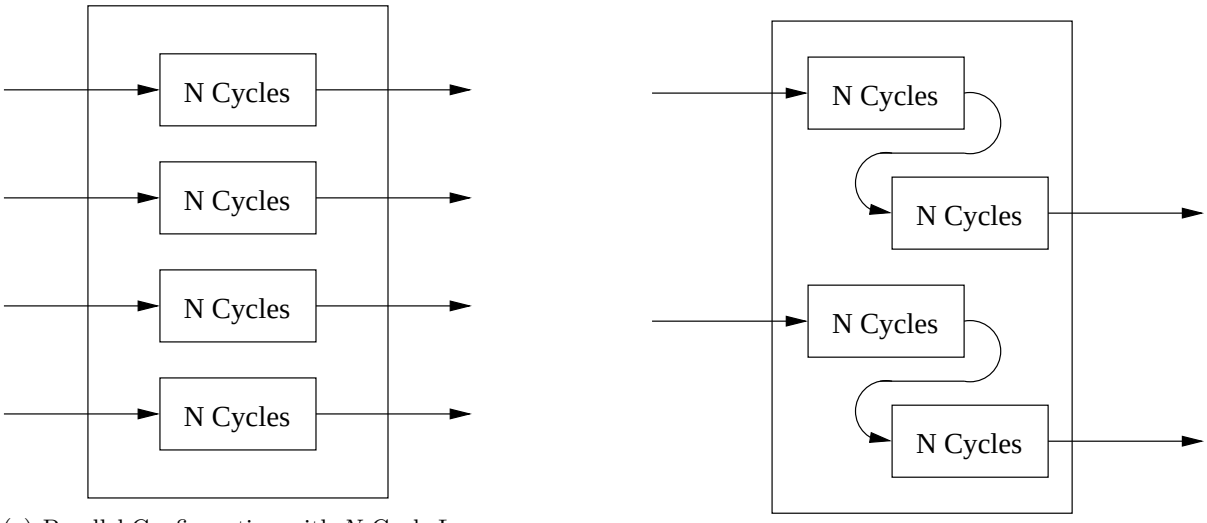
Figure 2.2: Concurrent Execution Between 8 Threads

ameliorate this cost by using multi-threaded processing engines. Any time a memory access is performed, the currently executing thread can perform a low-cost context switch to a thread that can perform useful work (as shown in Figure 2.2). These threads are hardware assisted, meaning that each thread has dedicated registers for storing its context information, so a context switch takes only a couple of processing cycles. By performing a context switch whenever the processing engine needs to wait for data, developers are able to take advantage of memory access latency to concurrently process other packets.

A network processor with multiple fast-path processing engines can simultaneously work on several packets, one in each engine. When each engine has multiple threads, even more packets can be processed concurrently. It is this parallel packet processing that allows a network processor to achieve the line speed processing required to handle today's network traffic. Hardware routers also distribute the workload using multiple processing engines, but on a different scale. For example, a CISCO 7000 series router [47], uses a Route/Switch Processor (RSP) for handling routing protocol tasks. The RSP manages the routing table and distributes it among the line cards that make up the router. The line cards handle data-packet routing based on the tables provided by the RSP and pass packets among themselves using a very high-speed switching backplane. This distributed switching is how hardware routers achieve their packet processing rates.

2.1.2 Trade-Offs

There are some trade-offs that occur when using multiple processing engines. For very simple processing, all of the processors can perform the same set of instructions on each packet. However, if the processing is more complicated, the processors can be put into a pipelined formation, with a processing engine performing a subset of the instructions on a packet before passing it to the next engine in line to execute a different set of instructions on the packet. The pipeline formation has the disadvantage of greatly increasing the latency that a packet experiences due to the number of instructions that are executed. For example, if each of the I processors performs the same N instructions on a packet (a parallel processing setup shown in Figure 2.3(a)), then the total processing latency that any packet experiences is N. However, in a pipelined setup of I processors (see Figure 2.3(b)), the latency becomes $I * N$. The more hops a packet makes, the more latency may occur. Packets also experience other types of latency during processing, for example, the latency experienced during memory accesses. A network processor may switch out a thread when a memory access is performed so the processing time is unchanged, but the packet processing is delayed while other threads on the processing engine continue executing. Also, in a pipelined configuration, packets are generally queued up waiting to be processed at the next microprocessor in the pipeline, which adds to the latency a packet experiences. This latency is not a significant problem in file transfers and similar traffic, or even in media streaming where the latency is only experienced at the very start. However, in real-time applications, such as VoIP, the total latency must be kept below a certain level to provide acceptable results. It is important to remember that



(a) Parallel Configuration with N Cycle Latency

(b) Pipelined Configuration with $2N$ Cycle Latency

Figure 2.3: Processor Configurations and Processing Latency

the actual packet processing rate is not affected by the processor configuration, as each processor is handling a different packet simultaneously, but the total latency a packet experiences does depend on the microprocessor configuration.

Even when the processors are configured in a parallel processing configuration, latency can be introduced by the need to synchronize memory accesses, since performing the same task may require accessing the same parts of memory. The need to synchronize memory accesses requires extra code, which reduces the packet processing rate. Also, scheduling must be implemented so that packets leave the router in the same order that they were received, which can require signals passed among the microprocessors. In this case, the packets experience more latency while waiting for the signals to arrive. In [52], it was found that the core processor alone, running at the same speed as a fast-path processor in the NPU, processed packets at nearly twice the rate compared to using the core processor and one fast-path processor in parallel, simply due to the synchronization required.

2.2 Network Processors

Each network processor differs in the number of fast path processors and the coprocessors available on each. These architectures show the differing views on what the key parts of packet processing are and how they can best be optimized.

2.2.1 Agere Payload Plus

The Agere PayloadPlus [46] series of network processors uses two hardware blocks to manage data path processing, the Fast Pattern Processor (FPP) and the Routing Switch Processor (RSP). The FPP contains a pattern processing engine to classify the packets and the RSP modifies packets and performs traffic management based on the results of the classification step. The PayloadPlus series appears to be the only network processor that is programmed using a functional programming language for classification tasks, which Agere claims is more efficient than the standard C-style languages used to program most NPUs [45].

2.2.2 IBM PowerNP Series

The IBM PowerNP [16] uses 16 microprocessors for handling fast-path packet processing tasks and a PowerPC processor for handling control-path processing. (The PowerNP and the Intel NPU are the only NPUs to include a core processor; the others use a processor located off-chip.) The 16 microprocessors are paired to create 8 Dyadic Protocol Processing Units (DPPU) and each DPPU manages 4 threads. The PowerNP Series uses several co-processors, including a co-processor to perform table lookups, one to calculate checksums and a co-processor for enqueuing data on the data port or switch fabric. The PowerNP also has a co-processor for performing policy management of flows and a classification co-processor, which can parse packet data before it is dispatched to a DPPU thread.

2.2.3 Motorola C-5 Series

Like the Intel IXP NPUs, the Motorola C-5 network processor [17] uses multiple microprocessors to handle the data streams. The unique architecture behind the C-5 is that the microprocessors are physically linked to a single data port, which means high congestion at one port does not affect the entire NPU. However, no matter how many microprocessors on the C-5, packets only travel through two, one at the ingress port and one at the egress port, which severely limits the amount of processing available to each packet. The Motorola C-5 also uses several co-processors that are available to all of the microprocessors. These co-processors include a processor for managing the switch fabric connection and off-loading the processing tasks required to send and receive data on the switch fabric. The C-5 also has a co-processor and special memory to assist with table lookups.

2.2.4 Intel IXP

The IXP NPU uses multiple RISC processors, called microengines (μEngines), to handle packets on the data path. These μEngines are also multi-threaded, as explained previously, and context switching is performed in a non-preemptive manner to facilitate programming. In addition, Intel uses a general purpose ARM processor to handle control-path packets. All of these processors are

contained on a single chip to speed packet processing. Similar to Figure 2.3, within the IXP chip, μEngines can be configured either in pipeline formation for larger processing tasks, in parallel to improve performance, or in a combination of the two configurations.

The Intel IXP also has several hardware blocks that act as coprocessors to aid in common computing tasks. Cyclic Redundancy Checks can be calculated without using the μEngines, hashes can be generated for various bit sizes, and there are hardware implemented clocks for generating timestamps. Additionally, second generation Intel IXP processors include a separate on-chip memory for fast accesses. This memory also includes hardware assisted queues to aid in passing packets among the μEngines.

The μEngines are quite complex as well. First of all, each μEngine has a local memory. This memory can be shared among the threads in the μEngine, or divided to give each thread a small number of private registers. In addition, there are sets of registers to store data written to or from the off-chip memory, which are used as buffers since reading and writing to the off-chip memory is significantly slower than the μEngine local registers.

2.3 Intel IXP2400

This thesis uses one of Intel's current generation of network processors, the IXP2400, which can support up to 4 Gbps of network traffic. The IXP2400 uses a 600 MHz Intel XScale (ARM V5STE compliant) core for slow-path processing. In addition, there are eight 600 MHz μEngines, each of which has 8 hardware-assisted threads. These 8 μEngines are arranged in 2 clusters of 4, each group having its own set of buses and memory controllers. Figure 2.4 shows the major hardware blocks of the IXP2400 and the connections among them. Intel also manufactures the IXP2800, which is essentially the IXP2400 with 16 μEngines and additional hardware coprocessors to facilitate cryptographic computations.

Each μEngine shown in Figure 2.5 has a 4KB instruction space, 640 words of local memory, 256 32-bit general purpose registers, and 256 registers for memory transfers to SRAM and DRAM, all of which are accessible by every thread on the μEngine. In addition, each μEngine has 128 Next-Neighbor (NN) registers. These NN registers are only accessible by the μEngine and its previous neighbor (by μEngine ID). These registers allow pipelined μEngines to synchronize or pass data without needing to use shared memory and the associated buses. Finally, every μEngine has a small 16 entry Content Addressable Memory (CAM) table. The CAM table can be accessed at the same speed as local memory, and is generally used to implement a cache.

Table 2.1 shows the different kinds of memory available on the IXP2400 and the number of cycles required to read or write to each type of memory. As usual, faster memory is more expensive, so there is less of the lower latency memory. The IXP2400 has 16KB of on-chip memory, called scratchpad memory. This memory is the lowest latency memory that can be accessed by all of the μEngines. In addition, this memory has special atomic bit operations, and 16 hardware assisted buffer rings with atomic put and get operations. Unfortunately, the

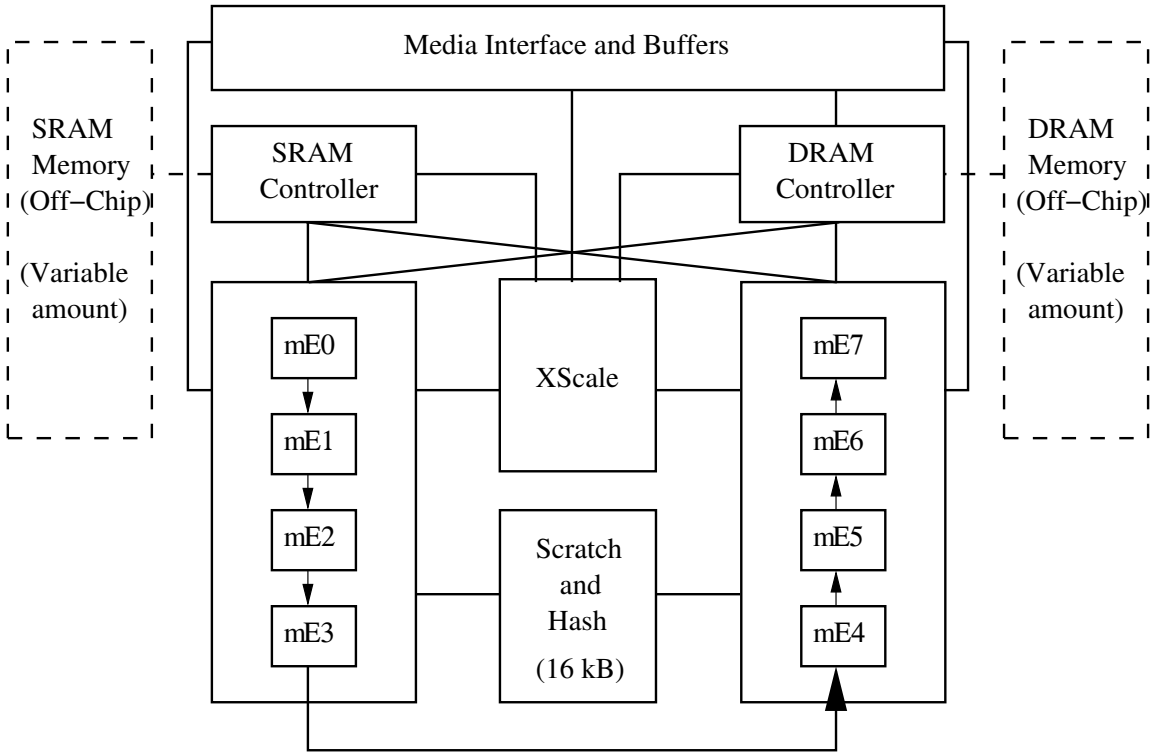


Figure 2.4: Block diagram of the Intel IXP2400 Network Processor

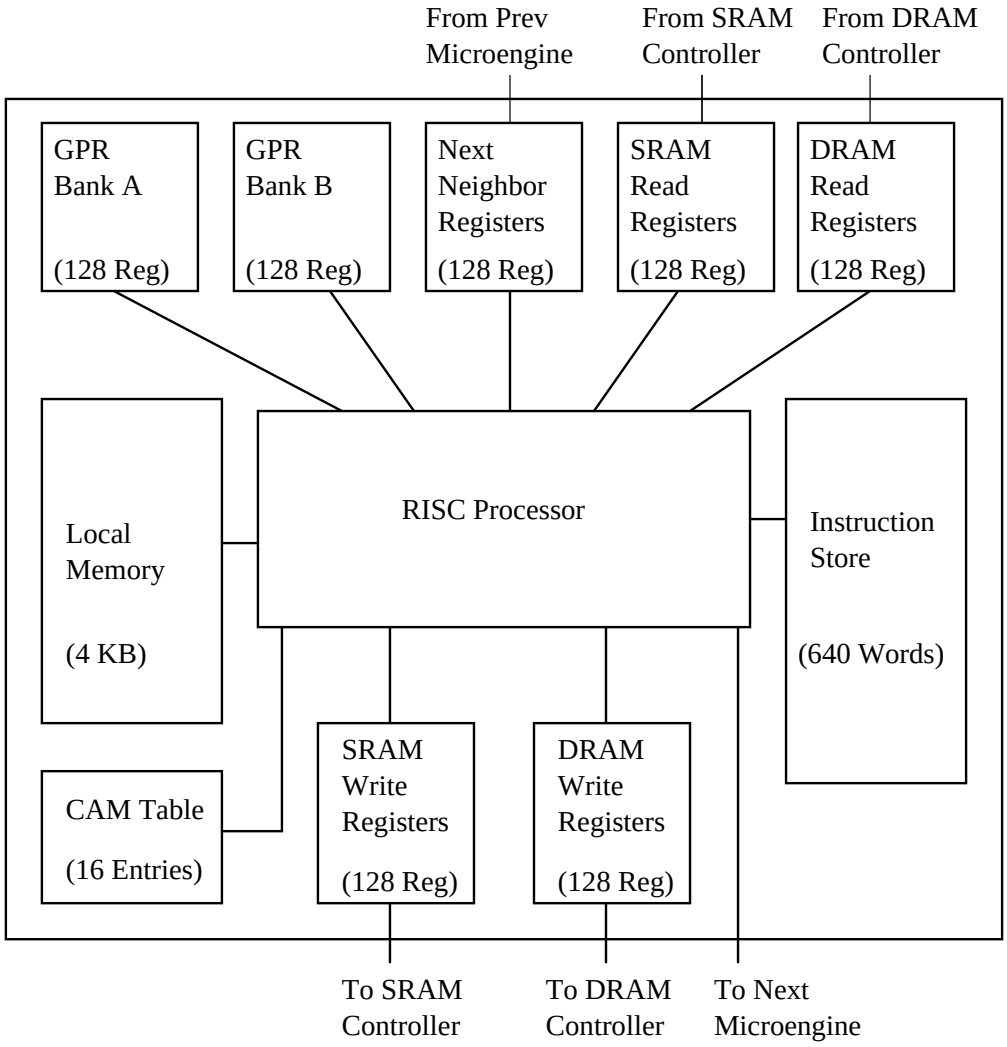


Figure 2.5: Block diagram of an IXP2400 μEngine

Table 2.1: Intel IXP2400 Memory Access Times

Memory Type	Access Time (Cycles)
Local Registers	3
Scratchpad	60
SRAM	90
DRAM	120

scratchpad memory resides on the hardware block that also contains the hash unit, which means that hash generation and scratch memory accesses can interfere with each other.

The fastest off-chip memory is the SRAM memory. The SRAM provides the same atomic operations as the scratchpad, but instead of a set of rings, each SRAM controller supports 64 queues with atomic enqueue and dequeue operations.

The slowest, but largest memory available to the IXP2400 is the DRAM. Access to DRAM is via a 64-bit bus, rather than the 32-bit bus of the other memory. Also, DRAM has a direct connection to the media interfaces, meaning that transfers can occur between the two units without going through a processor. This capability makes DRAM the obvious choice for storing the actual packets that the NPU handles.

Programmers of this system must choose where to store any particular piece of data, based on the frequency of accesses, amount of data being stored, and which of the two clusters the μEngine is in. For example, packets would likely be stored in DRAM because that data is infrequently accessed and very large, and because of the direct connection between the DRAM controller and the media interface. However, it would be unwise to put packet receiving and packet transmission functionality in the same μEngine cluster, because the frequency of calls would saturate the cluster's DRAM bus. By putting the receive functionality in one group, and transmission in the other, the load can be distributed more efficiently, since each cluster has its own DRAM bus. Similarly, it would be inefficient to put an IP lookup table in DRAM because of the high frequency of accesses. Such a table would best be stored in SRAM, or even in the scratch memory if the table is small enough.

Chapter 3

Related Work

This chapter explores a wide variety of topics related to network processors. It examines other coprocessors that perform tasks on behalf of the main processor, thereby offloading work from the processor and allowing hardware optimizations because of the more specific functionality of the coprocessor. Also, other processors are examined that encounter similar obstacles and solutions to network processors. Finally, a brief summary of existing applications using network processors is given as well as the highlights of current research being performed using network processors.

3.1 Related Hardware

3.1.1 Coprocessors

Coprocessors are used to reduce the load placed on a general processor. Historically, a math coprocessor is often used in computers to perform floating-point calculations on behalf of the main processor. The addition of this coprocessor greatly increased the speed of programs that performed many floating-point operations.

While floating-point operations have been incorporated into general processors for years, there is still a wide variety of coprocessors available, performing tasks from digital signal processing to cryptographical functions; several coprocessors exist for network related tasks. A network coprocessor can be included on ethernet cards for implementing the functions related to sending and receiving network packets, such as validating checksums, while more complex coprocessors can be used to take over all network-related tasks from a general processor [26]. A network processor is an evolutionary step above these coprocessors, designed to handle more complex communication-orientated tasks for an end host or router. With line speeds still increasing, researchers are now looking into using special coprocessors to aid NPUs. Using large CAM tables, these coprocessors could handle tasks such as classifying packets in a very small number of cycles [38] [36], saving the network processor cycles for modifying packets and performing complex scheduling computations.

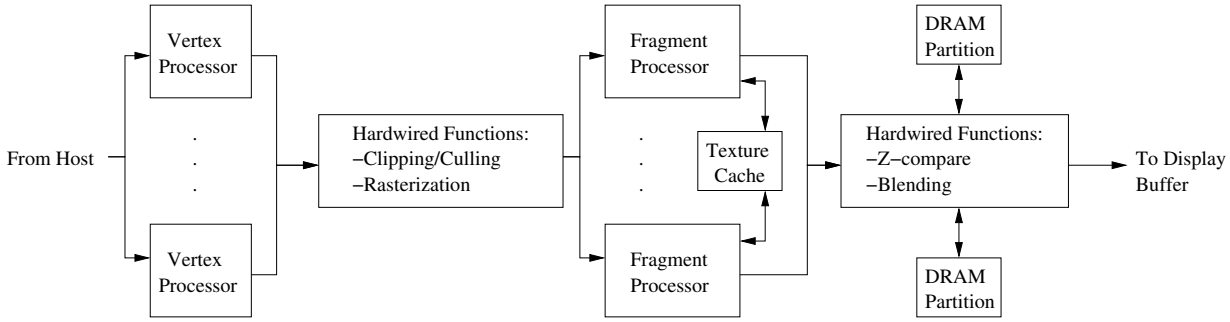


Figure 3.1: Block Diagram of the Multiple Processors in a GPU

In time, these coprocessors may be absorbed into an NPU, as floating-point coprocessors were absorbed into general-purpose processors.

3.1.2 Graphics Processors

Graphics Processing Units (GPUs), present on video cards for personal computers, are the most commonly used application-specific processor available. Specialized for performing the 2 and 3 dimensional calculations required for displaying data, these GPUs are often just as complex and difficult to program as network processors, and significant research efforts have been devoted to these challenges. Using the NVIDIA GeForce 6 series as an example, this section examines the typical architecture of these application-specific processors.

Graphics processors take data regarding object textures, movements, and vertices and apply a series of transformations to the data before displaying it on the screen (see Figure 3.1). First, the vertex processors take the objects' data and rotate, shift or scale the object according to commands from the main processor. The transformed data is sized to fit the screen and pixels that are within the viewing dimensions are grouped together based on where they are in the display. Fragment processors test these groups of pixel data to determine their location relative to other objects in the scene, then shading and textures are applied to the pixels based on those results. The final processing stage compares the depth of the objects being displayed and applies transparencies and color-blending. This procedure is a very computationally intense operation, and while not shown in Figure 3.1, multiple ASICs handling the blending operations. The pixel data is then sent to a buffer for display on the screen. A graphics card must be able to perform this sequence of operations fast enough to fill the buffer (which is the size of the screen) at least 24 times a second or the picture flickers at a rate that is noticeable to the human eye. As shown in Figure 3.1, this speed is achieved using multiple processors at each step that handle the pixel streams in parallel.

Each piece of data streamed though a GPU follows the same path and has the same operations

applied to it, similar to how each network packet follows the same path through an NPU. Also, both graphics and network processors handle the data streams in a pipeline fashion. Though GPU manufacturers are extremely secretive about their technology, some basic details are well-known and can be used to compare against other specialized processors. In general, the architectural details of a GPU are similar to an NPU.

First, graphics processors have a set of vertex processors, which take object-vertex data and transformation commands from the main processor and apply the transformations on the data as per user specifications. These vertex processors are essentially programmable floating-point processors and are quite analogous to the μEngines in the Intel IXP network processor, except the μEngines are also multi-threaded. The vertex processors are also designed to process massive amounts of data in parallel, like network processors handling packets.

Like network processors, GPUs must also handle latencies due to memory accesses. The fragment processors of the GeForce 6 series processor handle multiple sets of data simultaneously by performing the same instruction on multiple sets of data, a process known as SIMD (Single Instruction, Multiple Data). This helps to hide the latency of fetching data from the texture cache.

Also, like network processors, many graphics cards use consumer level memory to reduce costs. However, due to the speed requirements, the data bus on graphics cards is much faster than on network processor line cards. For example, the DRAM memory bus on an NVIDIA GeForce 6 series graphics card can transfer up to 35 gigabytes of data every second, while the memory bus on a Intel IXDP2400 development board is limited to 6 GB/sec [7] [8]. Graphics cards generally have fewer levels in their memory hierarchy, usually only local memory and DRAM, though the DRAM is partitioned into several independent sets to speed access. Unlike most network processors, the GeForce 6 series includes various types of caching memory for frequently accessed data, such as textures.

Programming Graphics Processors

Due to the rate of increase in processor capability, recent generations of graphics processors use programmable fragment and vertex processors. Initially, these processors were programmed using assembly languages, but recently higher-level languages have been developed for programming graphics processors. These new languages increase the portability of programs and facilitate incremental development, which is essential in graphical programs because of the real-time performance requirements.

The idea of programming graphics processors is attractive because of the amount of parallel processing available (see Figure 3.1) and the commodity price of GPUs. Indeed, graphics processors can be useful for many applications outside of displaying graphics. GPGPU (General-Purpose computation on GPUs) is an active research topic with numerous successes, particularly with database operations [24]. As GPUs are used to perform a larger variety of tasks and as the GPU architectures become more complicated, higher-level languages become extremely useful in

abstracting the specific details and idiosyncracies of the hardware and allowing programmers to be more productive in their efforts.

OpenGL [41] is a domain-specific language specification. The OpenGL compiler is tightly integrated with the graphics processor pipeline, creating an interface that is more intuitive for developers. Since OpenGL is a domain-specific language, it includes a larger set of domain-specific modules, such as lighting models, and can be more aggressive in optimizations involving the modules. OpenGL can also expose more graphics-orientated data to a developer, such as the Model view and World view matrices used in a GPU to describe the position of the viewing camera with respect to the world. DirectX [1], created by Microsoft, is a similar set of APIs as OpenGL.

Cg, or "C for graphics," is a high-level shading language developed by the graphics processor manufacturer NVIDIA [33]. A shading language instructs a graphics processor on how to shade pixels, so Cg primitives include data types that are commonly used in graphical calculations, such as 3 or 4 element vectors. Cg is designed to look and function similar to the C language, which is already familiar to most developers, while still being very similar to the actual assembly instructions on a graphics processor but abstracting enough of the architecture so graphics programming is easier. However, by making Cg sufficiently high-level and general, it is possible to program graphics processors for more general purposes, and easier to port programs written in the language between different GPUs.

To allow for portability between GPUs, Cg uses *profiles* to specify the capabilities of a graphics processor, and, hence which Cg operations it can perform. Profiles are a compromise between portability and abstracting away full hardware functionality. Profiles also allow for function overloading so that any special mechanisms on a particular processor can be used to optimize common operations. To port a program between processors requires applying a different profile, and only parts of the program that use specialized features of a processor are compromised. Moreover, to keep the Cg language more general, any extra capabilities of a GPU are exposed via libraries rather than through standard language operators.

Sh [34] is a C++ library that can be used to transform a set of Sh operations into code that runs on a GPU. The C++ code from outside the library is used as a framework for the structure of the shader code in order to allow developers to access the mechanisms (classes, operator overloading, etc) that are available in C++. Using this mechanism, it is also possible to exchange variables between the host application and the GPU code. This tight integration between the shader language and the host CPU language also facilitates optimization, since it is easier for a compiler to view the program as a whole.

Figure 3.2 shows the graphical languages detailed above and how they build on each other. There are also higher-level languages beyond those explained above. For example, Brook [4] is a language designed specifically for processing streams of data and can be compiled into a Cg program. In this figure, Brook would be at a level above Cg at the top of the hierarchy.

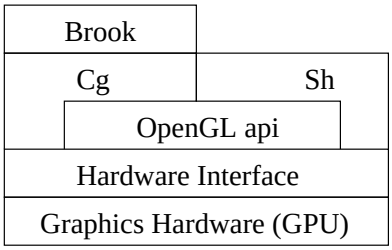


Figure 3.2: Interaction Layers of Graphics Programming Languages

3.1.3 Cell Broadband Engine (BE)

The Cell BE multiprocessor is being created as a joint venture between IBM, Sony and Toshiba, and will be powering Sony's third-generation PlayStation video game console and used in HDTVs to decode the data streams for display. The primary goals for the Cell are quite far-reaching, as it is designed to provide:

- outstanding performance on game consoles and multimedia devices
- quick responses to user actions and network behavior
- applicability to a wide range of platforms [15]

The design of the Cell is particularly interesting as the design challenges are very similar to that of a network processor. First of all, the responsiveness of the chip is a primary requirement for handling real-time multimedia data, just as it is in handling flows of packets at line speeds. Since it is intended for the next-generation of game consoles, the Cell must also handle Internet connections. More specifically, due to its use in multiplayer online game environments, the Cell is optimized to handle workloads related to communicating with multiple peers, very similar to an NPU. Finally, the Cell BE encounters the same latencies with regards to memory accesses and processor speed.

Because of similar obstacles, it is not surprising that this multiprocessor and network processors share many design points. First of all, the Cell processor contains multiple processing engines, similar to network processors. In the case of the Cell, these Synergistic Processor Elements (SPEs) [14] use the SIMD architecture that graphics processors also use. Each of these SPEs contain local memory, similarly to the μEngines on the Intel IXP. It is these SPEs that allows the Cell to decode streams of data quickly enough for users to watch streaming video. With Cell's multiple processor architecture, it would even be possible to include the chip in a server that can stream different HDTV streams to separate televisions.

The design goals of the compiler for the Cell multiprocessor, known as the Octopiler [13], are just as far-reaching as the goals for the hardware. In particular, the compiler must allow a developer to take full advantage of the SIMD architecture and multiple processors that are available on the Cell without requiring a developer to write completely specialized code for the hardware. The octopiler can generate code for both the SPEs and the core processor. Understanding the different processor architectures allows the compiler to better optimize the code by distributing the tasks among the processors.

One way that the octopiler is able to optimize applications is by taking scalar loops and combining the loops so that several iterations can be performed in parallel using a SIMD operation, a process called *auto-SIMDization* [50]. For example, the loop

```
for (int i=0; i<100; i++)
    a[i] = b[i] + c[i];
```

with 32 bit integer arrays `a`, `b` and `c` can be parallelized to perform 4 iterations at a time using the 128 bit registers in the SPEs and the contiguous data in the arrays. Auto-SIMDization also occurs when setting contiguous data in an object; the data is packed together into a single assignment operation.

In addition to SIMD operations, which affects a single SPE, the parallelism of multiple SPEs is also exploited. To execute tasks in parallel, regions of code are marked by programmers, and the octopiler manages the partitioning of this code among the SPEs and coordinating the execution.

3.1.4 The Success of Compilers and Higher Level Languages

There has always been a high demand for languages that help graphics developers be more productive, and hence release products, especially games, at a higher rate. Development companies often create customized rendering engines on top of the existing higher-level graphical languages to increase productivity for their type of applications. High-level graphics languages have succeeded for three reasons: the existence of standardized lower-level APIs (like OpenGL), their high demand and industry support, and because of the similarity in GPU architectures among different vendors.

The creation of the octopiler demonstrates that it is possible to create an efficient higher-level language for heterogenous multiprocessor systems. However, as with any compiler meant for a specialized hardware architecture, extending such a compiler to other architectures is much more difficult.

3.2 Network Processor Applications

Due to the programmability of a network processor, there is a wide variety of applications that can use network processors to handle packets, from simple firewalls and routers to streaming media

servers. Many of these applications use the previous generation of Intel network processors, the IXP1200. The IXP1200 uses a StrongARM core processor rather than the XScale streaming media servers. Many of these applications use this processor, which only has 6 (slower) μEngines compared to the 8 μEngines on the current generation Intel IXP2400.

3.2.1 Video Streaming

One rather unique application using the IXP1200 network processor is from the Osaka University in Japan. This application adjusts the quality of a video being streamed to handle connections of different speeds [52]. Video quality is adjusted by using a low-pass filter to remove high frequency data coefficients from the data blocks.

The most interesting aspect about this application is how the μEngines are used. First of all, this is one of the few applications where the μEngines all work on the same data packet. In addition, they are all slaved to the StrongARM core processor, which also does a large part of the packet processing (rather than just exception packet handling). For the video-quality adjustment application, the core processor receives a video packet and breaks it into a set of *video blocks*, storing the video blocks in shared memory for the μEngines. The μEngines then read out the individual video blocks and apply the low-pass filtering to them before copying them back to shared memory. Once all the video blocks in the packet have been filtered, the StrongARM processor repackages the video blocks and transmits the packet.

This application is an excellent example of what μEngines can be used for, as the low-pass filter is a relatively simple mathematical algorithm and well suited to the reduced instruction set. It also is a good example of what synchronization problems can occur when handling packets and using shared memory. In this case, all the μEngines are accessing the same packet simultaneously but each μEngine is accessing a different video block from the packet. The simultaneous access makes it important to manage memory accesses to eliminate duplicate work and to keep from overloading the data bus between memory and the μEngines. In fact, the developers found only a twofold increase in throughput when using all six μEngines and the StrongARM processors compared to just using the core processor to apply the filter. Using the StrongARM and a single μEngine actually resulted in slower throughput due to the added synchronization.

3.2.2 Firewalls

Firewalls sit at the edge of a network and filter packets. They fill an important role in network security, protecting networks from attacks and keeping the network bandwidth and computing resources available for processing useful packets. A firewall examines each packet and makes a forward/drop decision, after examining the packet header and performing some computation. The amount of computation performed for each packet determines the type of firewall. A basic *packet filter firewall* examines packet headers and makes a decision based on a set of simple rules (e.g., dropping all packets being sent to TCP port 21). A simple firewall can be implemented

on ASIC-based hardware to quickly process packets. However, a more complex firewall requires storing data and/or performing more complex computations, which cannot be easily implemented in hardware nor perform well on a slower general-purpose processor. Because of the increasing complexity of network attacks, most commercial firewalls are more complex than a basic packet-filter firewall. Most of these firewalls are *stateful firewalls*, meaning some state on previously received packets is stored and that state is used to make a forward/drop decision. The firewall in [29] performs a stateful packet inspection by storing the TCP state information in a hash table. The hash table is indexed on the 5-tuple packet header information with the hash being performed by the hash hardware block on the IXP2400 chip.

Recently, there has been a push towards the integration of network functionality into single devices. This integration can lower the cost of such devices and can also improve performance, as such devices can perform faster, since one function can leverage the work done by another function (e.g. sharing packet meta-data). In [49], the authors combine packet-filter firewall and Network Address Translation (NAT) functionalities into a single device. It was found that the primary bottleneck in the system is rule-lookup, as a fully configurable firewall must accept new rules and rule changes, looking up every rule for each packet. With only one μEngine handling firewall functionality, the estimated performance of the firewall is able to meet 1 Gigabit/sec line speeds when 1 rule lookup is necessary before a rule match is found. Allocating 3 μEngines of an IXP2400 to firewall functionality allowed an average of 10 rule lookups per packet for a firewall able to forward packets at a 1 Gigabit/sec line speed.

3.2.3 Software-Based Routers

Click [30] is a software architecture project developed at MIT for creating easily configurable routers. The primary idea is to divide a router into as many individual components as possible (e.g., packet classification, header verification, route lookup, etc). Each of these components is made into a separate Click *element* class. Elements communicate with each other via published interfaces, which can be used to call methods or access data. Click is a single-threaded program where the elements are scheduled on the router's CPU. Moving packets between elements causes these elements to be scheduled for execution on the CPU. The NP-Click project [42] works to address many of the problems that occur when applying Click to a network processor. The authors refer to these problems as the *implementation gap*, the difference between the high-level Click language and the lower-level implementation languages of network processors that must be used in order to properly use a network processor's specialized hardware and memory. To this end, NP-Click uses Click's idea of elements but gives the programmer a degree of control in specifying objects within an element and how elements communicate. A programmer allocates objects to different memory types based on the scope of the data, and controls how the processing elements map to the processing threads. Finally, an abstraction library is created to hide some of the hardware specifics of the different network processors. The end result of this project is an architecture that is able to abstract away much of the resource contention and specific hardware

concerns from a programmer. However, the programmer is still required to understand the network processor hardware to effectively use its available memory hierarchy and multi-threaded capabilities.

The NETKIT [9] project also uses the idea of software components with network processors. Here, components are stored in binary code compiled for the network processor's μEngines. These components are tightly-contained classes providing a complete piece of functionality. NETKIT components are very similar to JavaBeans [35], since both work towards the same goal: reusable code. With both JavaBeans and NETKIT, components all have well-defined interfaces for easy interchangeability, which keeps the modules separate and portable. Each NETKIT software component defines a set of *receptacles*, which must be linked to other existing components. In this way, NETKIT is able to provide some safety from improperly used components. As a further safety mechanism, NETKIT uses component frameworks to define rules and interfaces for a particular domain of components. These component frameworks even provide the property of nesting and inheritance. The NETKIT project looks at the idea of NPUs at a much more global scale than most other projects to create a structure that is easily managed and reconfigured during run-time and encompasses all the protocol layers.

3.2.4 Overlay Networks

Overlay networks are connections between end hosts that operate on top of an existing IP network. However, as these networks are implemented at the application rather than network level, packets experience longer processing delays because the routing operations are performed through user-level code. In [20], network processors are used between the end host and the network. The NPUs assist with managing the streams of data that are sent across the overlay network, while the end host does more computationally intense work on the data that requires more time and state. This work showcases network processors working at a higher protocol layer than previous examples. Here, in addition to handling IP level headers, the μEngines in the NPU must reassemble the data packets into application-level data units, which is much different than simply checking IP headers. Essentially, the attached network processor becomes the middleware for the networked application. Also, this research is one of the few examples where the μEngine code is changed while the NPU is running. With this network, the attached network processors store several different μEngine implementations. Should the load or the user's preferences change, the end host can send a message to the core processor of the NPU to switch to one of the other pre-existing stream handling implementations.

3.2.5 Grid Computing

Grid computing involves a large network of computers, each buying and selling spare processing cycles. Hosts looking to have remote computation performed on their behalf send the data and execution code to another computer, which executes the code on the data and returns the result.

To achieve this, a method is required for locating a remote host that is willing to execute the code. In [32], the authors use network processors to distribute computing requests. Computers willing to accept work register with a network processor, which stores their IP address and processing power in a special routing table. Hosts requiring remote computation send special *Task Message (TM)* packets with no IP destination, only the source address. The network processor recognizes these packets and forwards them to a registered computer with sufficient processing power to execute the code in the TM quickly. The result is sent back to the client using the IP source address in the TM packet. If the current network processor does not have any suitable computers registered, it passes the TM to another NPU deeper in the grid. In this manner, the grid is scalable in both the number of computers and the number of network processors running it.

3.2.6 Porting Existing Applications

There is also work being done to convert other existing network domain applications onto network processors. In [5], the authors have created a compiler that generates C-style (rather than assembler) code for the IXP1200 network processor from a `snort` [40] configuration file. The compiler creates a condition tree of packet header fields and values based on the configuration file and generates the NPU code to check for these values. While this may seem like a minor piece of research, porting an existing application to NPUs helps to show the performance improvements that can be made as well as provide NPU users with programs they are already familiar with.

3.3 Network Processor Research

The ForCES (Forwarding and Control Element Separation) IETF working group [25] is currently trying to develop a protocol that separates the control and forwarding planes of a network device. This protocol is a framework and set of mechanisms for combining the components of an IP device and standardizing the way data is passed between the control and data paths. By utilizing this protocol, improvements and innovations can be made to either processing plane without affecting the other.

The goals of the working group directly affect research efforts on network processors, as there is currently a tight integration between the two processing planes. Separating the planes would allow for easier changes to the hardware or software that implements the planes.

3.3.1 Programming Language Research

Language research is one of the most active areas involving network processors, primarily due to the difficulty in efficiently and effectively programming such a processor. This research involves not just creating libraries and components to develop a router, but also different application-specific languages to use when programming the packet processors.

In [21], the authors have designed a reconfigurable rule engine that executes on an NPU. The engine contains a table of domain-specific rules that are checked for each data event that the NPU receives. These rules determine the actions that the NPU receives, such as forwarding the event, updating some state, modifying the data, or creating a new event. Rules can be chained to perform more complex operations such as application-level routing. The rule engine is based on the SPLITS (*Software architecture for Programmable LIghtweighT Stream handling*) [19] architecture, which is a pipelined set of nodes that data events flow through. μEngines in an IXP2400 are implemented as *rule handlers*, which read the rule engine's decision table from either SRAM or DRAM. Each μEngine is implemented to handle a specific set of rules, with early classification determining which branch of rule handlers the event is sent to for processing.

Network Speed Technologies, Inc. has developed the Nova language [23] for network processors. The language exposes the specificities of the actual hardware (such as the communication facilities and hardware assisted queues), while hiding some of the more difficult aspects of the hardware. For example, Nova makes it much easier to extract or create protocol headers for packets using the *layout* data type. This meta-data type is an object containing a list of data fields in the layout and their sizes. The most useful aspect of this data type is that it can be populated with packet data using a single command, which means that if the protocol changes and new fields are added, it does not change the code except in the layout definition. This capability is particularly important as the insertion of a new data field can often require rewriting the extraction methods to ensure that the data is properly shifted and masked. This language primarily focuses on allocating registers, which is difficult as there are several different register banks of limited size, and spilling data over into different banks can be expensive (e.g., moving GPR data into local memory). Allocating registers is also made difficult as performing memory accesses, either read or write, usually require contiguous blocks of registers. Nova handles these problems using static analysis to identify unused data, ILP (Integer Linear Programming)-based optimization to allocate the variables, and coloring algorithms to handle register bank conflicts. In [22], a sample IPv4 forwarding application is developed using the μL language (a descendant of the Nova language) that allocates registers more efficiently than hand-coded assembly code (from the IXP2400 sample code provided by Intel) and arranges the code for optimal execution. As such, the μL forwarder performs more efficiently than the sample application.

Relatively few of the proposed languages model the concurrent properties of network processors. Generally, the languages focus on the domain specific operations that are necessary for packet processing, and leave the concurrent aspects of the work (synchronization and memory accesses) to the programmer. PacLang [11] is a linear-typed concurrent language for network processors. PacLang uses *thread molecules*, each of which contain a `main` method to define a thread in the packet processor. This language has yet to address debugging, optimization and maintenance, though it produces easily read code, which can ease maintenance tasks. PacLang is also able to allocate the threads to specific μEngines of a network processor, as detailed in Section 3.3.2 and [12]. In order to make the language very easy to understand and use, PacLang makes several assumptions about how the language is used. One assumption that PacLang makes

is that no two of the processing components are working on the same packet at any given time. There is no particular reason that this cannot be done on an NPU, so long as the tasks being performed are independent. However, there are many cases where this assumption does not hold, such as the video quality adjuster described in Section 3.2.1.

3.3.2 Task Partitioning

Allocating tasks to separate threads or processing engines has been heavily researched in real-time and distributed systems. Particular care must be taken when allocating tasks to different processors, as load balancing and pipelining must be taken into consideration. Much of this research work focuses on creating compilers or other software tools that automatically perform these operations for the user. One of the primary goals of these tools is to minimize communication among massively parallel systems. This issue is less of a problem with network processors, as the hardware is specialized for communicating among the microprocessor components of the NPUs and because of the much smaller number of microprocessor components. However, there are a set of critical constraints that are faced when dividing tasks among microprocessors such as the μEngines on an Intel NPU:

- tasks can take different amounts of execution time
- communication links between tasks can take different lengths of time
- communication links are shared, so tasks must compete for access [31]

Much of the existing research only looks at mapping the software to multiple processors rather than splitting or combining computational blocks to better fit the processors as a group. This capability is particularly important to avoid bottlenecks that would slow processing and reduce the throughput of a packet processor.

Task partitioning is also a heavily researched topic outside the domain of network communications; much of the research is based on efforts to improve hardware design and cost. In these cases, the general goal is to lower the total system cost. As mentioned in Chapter 1, ASICs are more expensive to design and produce, but are significantly faster than off-the-shelf processors. Therefore, the more work that can be done without ASICs, the cheaper the product.

For example, digital signal processing systems also face many of the same difficulties as network processors. Both have high performance requirements that must be balanced with low production cost. In [2], the authors have developed an algorithm that takes a system specification and creates an allocation of the system components, a partitioning of the functionality, and a pipelined design minimizing the number of ASICs required, all while maintaining the required performance. The algorithm works by taking a control flow graph (CFG) of tasks and allocating them to either hardware or software systems, based on the time it would take to execute the task. Then, for each software allocated task, the algorithm chooses the cheapest processor that

can execute the task at the required speed. By allocating as many tasks as possible to software rather than specialized hardware, the total system cost is lowered.

Recent developments in wireless communications has lead to much larger data streams that require rapid processing. As cell phones become more like multimedia broadband devices, wireless receivers must be able to handle data speeds at rates of tens of Megabits, rather than Kilobits. To this end, [39] looks at using multiple DSPs and a more parallelized processing algorithm to provide faster communication to meet users' new requirements. Their work is facilitated in part by the algorithms that are used in most current devices, which use matrices to aid in determining the interference a user is experiencing. Since matrix operations are inherently parallel, the researchers are able to easily partition the matrix data among the array of processors. As opposed to [2], this paper focuses on parallel hardware computation rather than pipelined computation.

One research project that does work on splitting a piece of code to work on multiple pipelined processors is the PacLang project. As mentioned earlier, PacLang is a high-level, domain specific programming language developed specifically for network processors. This language uses tasks as the basic objects and queues for communicating among tasks. Tasks are executed repeatedly and concurrently in this language, exactly as they would if implemented on a thread in a μEngine. An Architecture Mapping Script (AMS) is used to define the transformations of the architecture neutral PacLang code to machine code to be executed on the target hardware, which in the paper is the Intel IXP2400.

Currently, PacLang task partitioning only supports 3 operations: splitting a task, combining two tasks into one and multiplexing queues [12]. Combinations of these operations are used to create the final application. Unfortunately, PacLang is still missing some important functionality. For example, the programmer is still required to define where objects are stored in memory.

The Fairly Fast Packet Filter Programming Language (FPL-3) [10] is a packet filtering language that allows the programmer to distribute packet processing not just across μEngines within a network processor, but also across network processors. NPUs are arranged in a tree structure, with each node in a tree splitting the data (based on the header or payload contents) among the node's children. This structure means that each node handles fewer packets, so more processing can be performed on each. As a result, much more sophisticated applications, such as intrusion or spam detection, can be run. FPL-3 also supports basic load balancing to allow for scalable distributed processing. The processing state is communicated among tree nodes encoding information in the ethernet header of the packet, as only 2 bytes of 6 are used for the actual address of the next node.

Chapter 4

Software Development

4.1 Developing for the IXP2400

4.1.1 IXA Portability Framework and Dispatch Loops

Programming at the hardware layer requires a very good understanding of the hardware and all of its idiosyncracies. In general, the resulting programs are difficult or impossible to transfer to other hardware or include as components in another application. To mitigate these issues, Intel has created the Internet Exchange Architecture (IXA) Portability framework for packet processing applications. This framework is a library of basic actions, such as receiving a packet or removing the ethernet header from a received packet. The basic unit in the framework is a *microblock*, which is a functional unit of processing. The programmer joins multiple microblocks together to form a pipeline to process a packet. For example, a simple packet processing system would only have 3 sets of microblocks in a pipeline: receive, process and transmit microblocks. Packets would be passed from one microblock to the next. The series of packet processing tasks are placed in a *dispatch loop*, which loops once per packet. An example of a general dispatch loop for processing a received packet is shown in Figure 4.1 and a further discussion of the IXA Portability Framework can be found in [6]. The dispatch loop also associates commonly accessed packet-related information (such as the packet length) with special data structures that the dispatch loop and library methods access.

Also, to allow the programmer to define the data transfer mechanisms among μEngines, all of the IXA library microblocks call abstract methods to transfer data, following a design pattern typically known as *template methods* [18]. An example of one of these template methods is the `dl_rx()` method in Figure 4.2, which is called by the receive microblock to transfer received packet data to another μEngine for processing. Similarly, the template method `dl_tx()` is called by the transmit microblock to get packets to transmit. Template methods like `dl_rx()` and `dl_tx()` are defined by the programmer and are considered part of the dispatch loop. Figure 4.2 show the relationship between the template methods and the dispatch loop. In the figure, the

```
dispatch_loop:
    dl_source()         //gets a packet
    process()           //processes it using functions from the IXA
                        //Framework library
    dl_sink()           //passes it on
    br(dispatch_loop)   //repeat loop for the next packet
```

Figure 4.1: Dispatch Loop Code Example

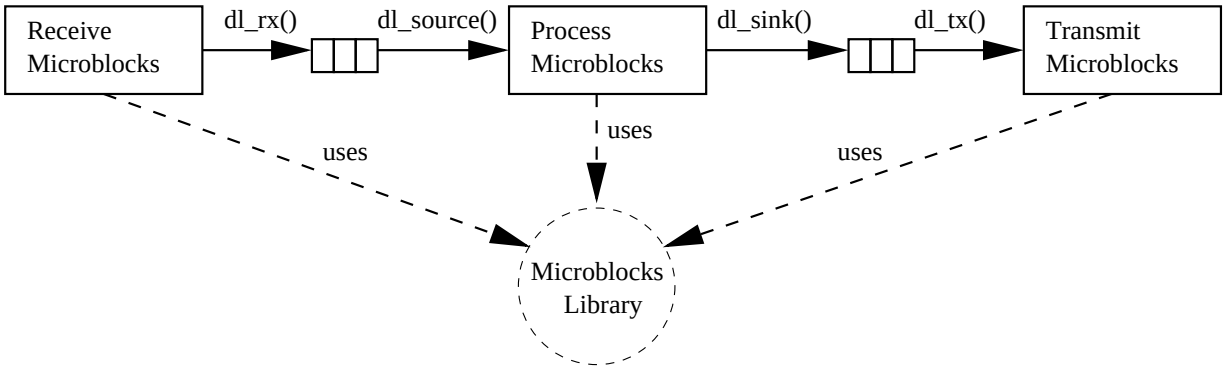


Figure 4.2: Dispatch Loop Block Example

receive microblock uses the `dl_rx()` method to place received packets on a queue and the process microblock uses the `dl_source()` method to get the received packets from the queue. Similarly, the `dl_sink()` method is used by the process microblocks to place the packets on a queue and the transmit microblocks use the `dl_tx()` method to get the packets for transmission.

In the case of the IXPs and their μEngines, one or more microblocks is assigned to a μEngine. The connections for transferring packets are through one of the many available data transfer mechanisms, such as Next Neighbor registers, scratch memory rings, or other forms of shared memory. In the cases where there are many more microblocks than μEngines, it becomes necessary to allocate multiple microblocks to a single μEngine, which is why the connections are all programmer-specified. In this case, microblocks can be assigned to different threads on the μEngine, and when executed is similar to coroutines passing control among themselves. Otherwise, each μEngine thread runs consecutively through all of the microblocks, only releasing control when a memory access is performed.

As with any function library, there are advantages and disadvantages to the Intel framework. Given that the language is specific to the hardware, it only has a limited use on other manufacturers' NPUs. However, this framework does allow the code to be more easily shifted among

different Intel IXP hardware. For example, it would be relatively easy to port an application from the IXP2400 to the IXP2855, which uses the same technology but adds hardware blocks for faster cryptographic calculations and some encoding/decoding functionality to support security using the specialized cryptographical units on the IXP2855.

4.1.2 Programming Languages

Intel provides 2 languages for programming the IXP line of network processors. The first language is an μEngine assembly language that directly translates to IXP machine code, that is, one line of code translates to one operation on the chip. This language also contains several basic control structures, such as while loops and if statements, to ease programming efforts. This language does not use an execution stack, however, the language provides the capability of functions using *macros*. In the μEngine assembly language, macros are functions that are inserted inline into the code when it is compiled. Macros allow programmers to create functions to perform tasks that are executed often, and to modularize the code so that it can be maintained with greater ease.

Intel also provides a C-style programming language that compiles into the IXP assembly language. This language gives the programmer access to the typical functionality found in ANSI C, such as type safety, structs, arrays and pointers. Since there no execution stack, methods can be either inlined during compilation or compiled as subroutines accessed by the `jump(address)` and `rtn(address)` commands. Since many of the specialized hardware options are unavailable using the ANSI C language, Intel's language also includes *intrinsic* commands, which translates directly into known sets of assembly code. For example, the intrinsic command

```
void sram_test_and_incr(volatile void __declspec(sram) *address,
                        __declspec(sram_write_reg) unsigned int *val,
                        sync_t sync,
                        SIGNAL *sig)
```

translates to the following assembly macros and instructions:

```
if (sync_t == sig_done)
    sram[test_and_incr, $val, address, 0], sig_done[*sig]
if (sync_t == ctx_swap)
    sram[test_and_incr, $val, address, 0], ctx_swap[*sig]
```

In addition to intrinsics, the language allows the programmer to insert assembly instructions into the code using the `__asm{...}` tag.

Higher-level languages typically include more control structures and mechanisms to speed up programming efforts, but these mechanisms can create programs that run slower than an equivalent assembly language version. Highly optimized compilers can improve the performance for high level languages, but it is still a common practice to code the key methods of an application

in assembly and code the rest in C. Applying this practice to network processors, the dispatch loop would be coded in C and the microblocks in assembly.

The scheduler in this thesis is coded using the IXP assembly language. The main reason for this choice is that in order to be efficient, a packet scheduler must be very simple, hence it can be transformed into the assembly language very easily. Additionally, since each line of assembly translates to a single μEngine cycle, static analysis based on lines of code is feasible.

4.1.3 Thread Arbitration

The μEngine arbiter is responsible for determining which thread is going to execute next. The arbiter is fixed and uses a strict round-robin arbitration scheme. Within a μEngine, each thread is numbered based on the registers that stores the thread's context. The round-robin scheme cycles continuously through these contexts. The arbiter is constantly running, checking the local Control Status Registers (CSRs). When a thread performs a context switch, it sets the `wake_up_events` CSR indicating what signal or condition (explained in Section 4.2.1) must be set in order for the thread to continue execution. The arbiter continuously compares the `wake_up_events` registers to the `event_signals` CSR, which shows what signals have been activated. This way, after a context switch is performed, the arbiter is instantly able to determine the next thread to execute and activate it.

Without the arbiter, a thread would have to check the CSRs for itself to determine the next thread to execute. Even though each thread knows what code every other thread on the μEngine is executing, it is impossible for a thread to know the exact status of another thread. This is because memory accesses take a variable amount of time, depending on what other accesses are being performed at the same time. Each μEngine can perform memory accesses at the same time, so it is impossible to determine in what order the accesses occur and how long it takes before the result is available. As a result, a pure coroutine style of arbitration with one thread choosing which thread to pass control to is not possible, because there is no guarantee the other thread is able to execute.

4.2 Synchronization Within a μEngine

The IXP2400 NPU has 8 μEngines, each of which has 8 hardware assisted threads. This structure means a programmer must handle up to 64 threads, 8 of which (one per μEngine) are running simultaneously. As a result, relationships among μEngines have to be carefully managed, as most of the memory is shared among all of them. To this end, the IXP hardware assembler language includes many built-in synchronization mechanisms, which are detailed in the following sections.

The primary reason for the multiple threads in each μEngine is to allow it to handle multiple packets concurrently. The idea is for a thread to suspend whenever it performs an operation where the thread must wait for a result. At that point, a thread that can execute is scheduled by the arbiter until it too executes a delay-causing operation. For example, a context switch is

usually performed whenever a memory access occurs. This context switching allows the μEngine to be idle for fewer processing cycles, resulting in a higher rate of packet processing. The need to perform context switches is one of the main reasons that a programmer must concern themselves with concurrency, the other being multiple μEngines working in parallel on the same task.

The arbitration method for activated threads does make handling synchronization somewhat easier. Within a μEngine, threads are marked as inactive, executing, or ready to execute. The arbiter stores this information and when one thread gives up control, the arbiter gives control to the next highest thread number that is ready to execute in a round-robin fashion. This specific thread arbitration makes it much simpler to set an order for packet processing. Arbitration is also made simpler due to the non-preemptive nature of the threads. Since threads do not lose control unless they explicitly release it, certain problems cannot occur. For example, critical sections rarely occur, as the programmer can guarantee that no other threads enter the critical section simply by not releasing control. However, there are some cases where this is impossible. For example, if a memory access is made within the critical section, a thread either gives up control or else spins, wasting processing cycles. Even in this case, locking becomes a much simpler task, as it can be done using variables that are shared among the μEngines threads, where the non-preemptive nature of the threads again means that it is impossible for 2 threads to be testing and setting the lock (which is not an atomic operation on μEngine registers) at the same time.

4.2.1 Signals

Where synchronization is required, the two IXP programming languages provide data types called signals. In these programming languages, a signal is essentially a condition variable that the compiler associates with a user-defined variable. Signals are primarily used when a thread has to wait for a memory access to complete, and wants to block to avoid wasting processing cycles. To do this, it associates a signal variable to the memory access and sets a control status register indicating that the thread is not to be woken up until that signal has been flagged. Then the thread performs a context switch. The μEngine arbiter monitors the outstanding signals and what signals are required to activate a thread, and re-activates the thread when all of the required signals have been flagged. Each μEngine is able to wait on up to 15 signals at any given time and these available signals are shared among the threads on the μEngine.

Signals can also be sent among the threads of the μEngines using the local Control Status Registers (CSRs). In these cases, the CSRs can be set to send the signal to any given thread ID on the μEngine, as well as the next or previous contexts, based on the current thread's ID. Being able to send to the next μEngine is particularly useful when trying to order packets as they are being processed.

```
loop:
    cam_lookup(memory_addr, cam_return)     //search for the lock
    if (CAM_MISS)                   //lock not found
        get_cam_lru_entry(lru)      //try to insert a new lock into
        if (lru.state == UNLOCKED)  //least recently used entry
            cam_insert(memory_addr, lru)
        else
            context_switch()        //no available unlocked entries
            goto loop               //try again
    else if (CAM_HIT)
        if (cam_return.state == UNLOCKED)
            set_cam_state(cam_return, LOCKED)   //acquire lock
        else
            context_switch()
            goto_loop

execute_critical_section()

set_cam_state(cam_return, UNLOCKED)     //release lock
```

Figure 4.3: Pseudocode for a CAM table locking mechanism

4.2.2 Content Addressable Memory

Another method for synchronizing threads within a μEngine is to use the Content Addressable Memory (CAM) unit. The CAM unit allows for fast access to 16 32-bit entries, each of which has 4 bits to store state. In [27], the authors describe a way the CAM entries can each represent a memory address, or some other unique identifier, and the state of the CAM entry can represent whether or not a lock is open. This technique allows the CAM entry to support 16 unique items being locked at a time, with any piece of data not in the CAM table considered unlocked. To acquire a lock on an object in memory, a thread performs a CAM lookup on that memory address. If the memory address is in the CAM table, the state of the entry, either locked or unlocked, is returned. If no entry is returned, then a new lock is inserted into the CAM table, removing an unused lock, to lock the data. Figure 4.3 shows the pseudocode for using the CAM table as a lock.

This synchronization method is only possible given the non-preemptive nature of the threads; otherwise, it would be possible for another thread to interrupt the CAM lookup, perhaps stealing the lock from another thread. The primary advantage of CAM table locks is that it is possible to query all 16 entries simultaneously. Also, since the CAM unit is local to the μEngine, this results in better performance when a thread has to poll to acquire a lock. The major drawbacks to this synchronization mechanism are the small size of the CAM table and that the CAM table may be more useful as a small cache.

4.3 Synchronization Among μEngines

Synchronization among μEngines is more difficult, particularly when accessing shared memory. In these cases, both synchronization and critical sections may be required. Again, the hardware provides some mechanisms to aid the programmer in both controlling memory accesses and in ordering packet processing among μEngines.

4.3.1 Signals

Signals can be used among μEngines in a similar manner as they are used among context threads within a μEngine. In this case, the IXP languages allow the programmer to signal the next or previous neighbor, which is useful when ordering thread execution. Figure 4.4 shows an example of how signals can be used within threads and among μEngines to order packet processing. In this diagram, μEngines i and j are used to perform IP layer processing, which is an intensive task and has been replicated to 2 μEngines to remove a possible bottleneck in packet forwarding. To ensure that packets leave the forwarder in the same order as received, a signal is passed between each thread in a μEngine, then onto the next μEngine. In this example, the signal is treated like a token; once the signal is received, the μEngine is allowed to pass the current packet to the

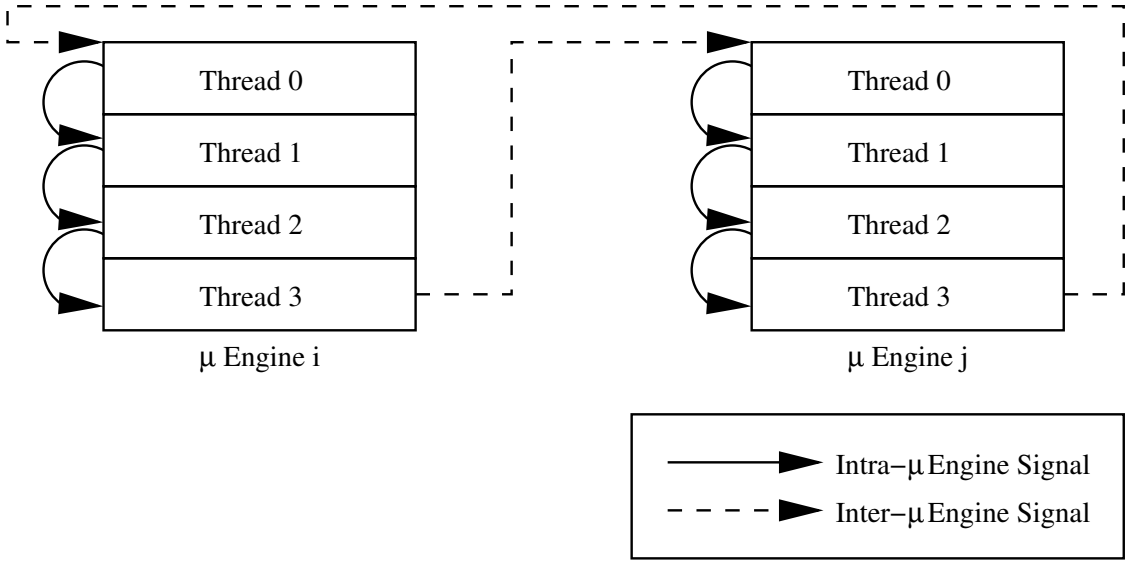


Figure 4.4: Signalling within and among μEngines to ensure ordered packet transmission

next processing stage and begin processing a new packet. This structure allows each μEngine to perform the IP processing simultaneously and only wait at the final step.

It is also possible for the threads to signal a particular thread on a different μEngine. This capability is done using a Control Status Register located with the Scratchpad, Hash and CAP (SHaC) unit, which is outside of all the μEngines. Hence, these signals take longer to process than signals local to a μEngine.

4.3.2 Shared Memory

Signals can help with ordering threads among μEngines, but are only useful in synchronization tasks. To this end, the various memory controllers on the IXP2400 NPU provide the following mechanisms to aid in creating mutual exclusion locks that can be shared among μEngines.

Scratch Memory

The scratch memory unit provides a set of atomic operations (e.g., test_and_set, test_and_incr) which can be used to create a lock accessible to multiple μEngines. Since this memory is large, there are many locks available, which allows for greater scalability compared to the locking mechanisms available within a μEngine. The atomic operations each return the previous value that the memory held, meaning that to create a lock, the programmer calls `test_and_set` to set the value of the memory to 1, and checks the return value as to whether or not it was 1 before

```
loop:
    test_and_set(lock_addr, lock_val, 1)
    if (lock_addr == 1)     //check the lock's previous value
        context_switch()    //lock is being held by another thread
        goto loop           //so swap out, then try again

execute_critical_section()       //have the lock now

set(lock_addr, 0)           //release the lock
```

Figure 4.5: Pseudocode for a critical section lock using atomic memory operations

the call, as shown in Figure 4.5. This method has some disadvantages; the primary one being that any thread attempting to get the lock must poll for it, which is slow because it requires the thread to wait for memory that is outside the μEngine and allows for the possibility of starvation. Even though the thread performs a context switch each time the lock is polled, it still puts more load on the data bus. However,

The scratch memory unit also provides 16 circular data-rings with atomic get and put operations. Using these rings, it is possible to create consumers and producers with safe access to these rings. For example, going back to Figure 4.4, a scratch ring with pointers to packets in memory could be filled by the receiving threads in a forwarder program, and then emptied by the μEngines performing route lookups. In turn, once these routing lookups are complete, the μEngines could insert the memory addresses into another data ring where they would be removed by the code responsible for the next stage of processing.

SRAM

The SRAM memory controller provides the same atomic operations as scratch memory. The primary difference here is that SRAM accesses take more cycles to complete compared to scratch memory accesses but SRAM is much larger than scratch memory, allowing for more locks.

Where the scratch memory provides rings, each SRAM memory controller provides special memory table for queues. Each table has space for 64 queues with atomic get and put operations. However, it is possible to remove one queue from the memory table in order to use another. A queue descriptor containing the head and tail pointers of the queue is stored in SRAM. Queues can be loaded into one of the 64 available controller slots by copying the queue descriptor to that slot. Similarly, a queue can be unloaded from a slot (e.g., to make room for loading a different queue) by copying the queue descriptor back into SRAM. In this way, SRAM is able to support a dynamic number of queues of dynamic size, where the limits are determined by the amount of free memory. This capability is very useful for complex schedulers or queue managers, as the 16

rings provided by scratch memory are likely too few and too small to handle large volumes of packets and flows.

Next Neighbor Registers

The Next Neighbor registers provide a direct connection between one μEngine and registers in the next highest numbered μEngine. This connection is maintained in hardware as a circular buffer, with the μEngine that owns the registers able to remove from the buffer, and only the previous numbered μEngine able to add data to it. This structure creates a producer/consumer relationship between the μEngines, and allows synchronization of the data pipeline between them. This method of synchronization is the fastest method of communicating data between μEngines due to the direct data bus between them. However, it is limited by the number of Next Neighbor registers on the μEngine and by the fact that the connection can only allow for a single consumer (the current μEngine) and a single producer (the previous μEngine).

4.4 Synchronization Between μEngines and the XScale Processor

The XScale core processor may need to communicate with the μEngines of the network processor. In this thesis, the XScale processor is only used to initialize the μEngines and no subsequent synchronization is required with the core processor; however, the possible methods of synchronization are included for completeness.

All communication from the XScale core processor is done using the processor's logical memory space, which covers all of the shared memory (scratch, SRAM and DRAM memory) within the NPU. This logical space also covers a small subset of the μEngine registers, allowing for direct communication between the two sets of processors.

4.4.1 Shared Memory

The XScale core processor has access to all of the memory that is not local to the μEngines. Memory that is only used by the core processor is allocated in DRAM. Access to memory that is shared with the μEngines is granted via a Resource Manager, a library which handles memory that a standard C memory manager does not have access to (i.e., SRAM and scratch memory). Programs running in the core processor (called *core components*) specify both the memory type and channel when performing any memory allocation using the Resource Manager. The XScale sees all of the NPU memory as a single logical space and must use the Resource Manager to translate any memory pointers into ones that are correct addresses for the μEngines. In this way, the XScale is able to allocate and set data for the μEngines to use. The XScale core processor can use the same set of atomic operations (such as `test_and_set`) that are available to the μEngines, and also has a set of APIs for acquiring and releasing locks in shared memory to facilitate coordination with the μEngines.

4.4.2 Rings and Queues

Similar to shared memory, the XScale processor has access to all the rings and queues in scratch and SRAM memory within its logical memory space. In this manner, it is possible to set up consumer/producer relationships among the core processor and the μEngines. This method of communication is commonly used as μEngines send exception packets (e.g., those that need more complex processing) to the XScale for processing, and then the XScale puts the packets back onto a ring or queue for the μEngine to transmit.

4.4.3 Variables and Registers

The XScale is able to set constant variables in μEngine programs when they are loaded into the μEngine instruction store. This capability allows the XScale to allocate and configure tables for the μEngines and then integrate the memory address of the data structures into the actual code.

The core processor is also able to read and write to the transfer registers of the μEngines. This capability allows for faster communication between the XScale and a single μEngine, as both processors do not need to make memory accesses.

4.4.4 Signals

The XScale also has access to the CSR registers local to each μEngine, which allows the core processor to set signals within a μEngine. It is through these registers that the XScale starts and stops a μEngine.

Chapter 5

Scheduler Architecture

This chapter introduces the scheduler implemented in this thesis, as well as detailing adaptations made to simplify it. Next, the primary data structures used in the scheduler are described before finally examining how this scheduler is mapped onto the μEngines of the IXP2400.

5.1 Packet Scheduling

5.1.1 A General Processor Sharing Scheduler

Many packet schedulers work by emulating the function of a General Processor Sharing (GPS) server. A GPS server is explained in [37] as being a work-conserving device that is able to handle multiple packet streams simultaneously and that assumes traffic to be infinitely divisible. The GPS server sends out bits of each packet according to the relative amount of service (termed weight) that each packet flow needs to receive. Streams with higher service rates are able to transmit larger volumes of data over a given period of time, assuming that the stream is producing sufficient amounts of traffic. A GPS scheduler is an *ideal* scheduler in that it exhibits guaranteed fairness among flows, meaning that each packet stream is serviced using a proportion of the total bandwidth, based on the weight assigned to the stream. Also, since the bandwidth allocated to a flow is the guaranteed minimum bandwidth that the flow receives, it is possible to bound the queueing delay at the server without having to consider any of the other streams at the scheduler. Unfortunately, implementing a GPS scheduler is impossible, since packets are not infinitely divisible and must be transmitted as whole objects. Nevertheless, GPS and its associated properties are used as a benchmark with which to measure other schedulers.

5.1.2 Timestamp Schedulers

One common method for emulating a GPS scheduler is to use the concept of *virtual time*. Virtual time allows the scheduler to use the time at which packets arrive to calculate a deadline for

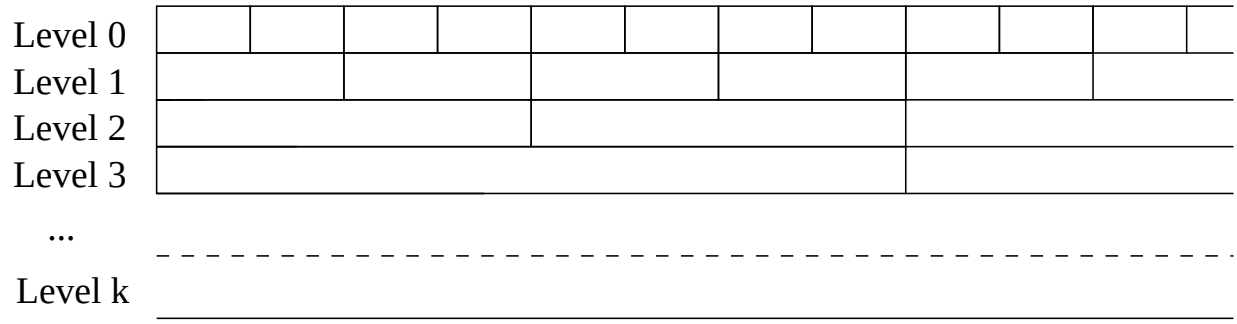


Figure 5.1: Stratified Timer Wheels

when the scheduler should transmit them. In this way, packet handling can be prioritized using a packet's virtual transmit (or finish) time, achieving performance that is similar to a GPS scheduler, but different in execution by transmitting an entire packet at a time rather than parts of packets. However, this solution can only be an emulation of a GPS scheduler, because a small packet that has a high service rate associated with it can arrive at a scheduler while it is transmitting a large packet. Since the scheduler must complete its transmission of the large packet, the smaller one (which should have been sent out first, having a lower virtual finish time) misses its virtual transmission deadline.

One example of a timestamp scheduler is the Virtual Clock Scheduler [53], where each packet in a data stream receives a virtual finish time based on its reserved rate and the finish time of the previous packet in the stream (i.e., for a new packet at queue position i, $Finish_i = Finish_{i-1} + \frac{packet\_length}{reserved\_rate}$). Packets are handled in order based on the assigned finish time. The Virtual Clock scheduler suffers from the problem with small, high service rate packets mentioned above and can not guarantee fairness comparable to a GPS scheduler. Achieving the fairness properties of a GPS scheduler usually requires a non-constant execution time [53] [44], which results in a scheduler that does not scale properly with increases in data transmission rates.

5.1.3 SI-WF^{2}Q: A Fair Timestamp Scheduler with Constant Execution

This thesis describes an implementation of an existing scheduling algorithm, SI-WF^{2}Q, which is able to achieve fair-weighted packet scheduling in constant time and with low constant factors [28].

Packets arriving at the router are grouped into data flows based on their source and destination addresses. In addition to a encapsulating a queue of packets, these flows will also have some metadata associated with them, for example, the start time of the first packet in the flow and the service rate reserved for the flow. The higher the service rate, the earlier packets are transmitted towards the destination. This scheduler uses relative rates, with all services rates being expressed

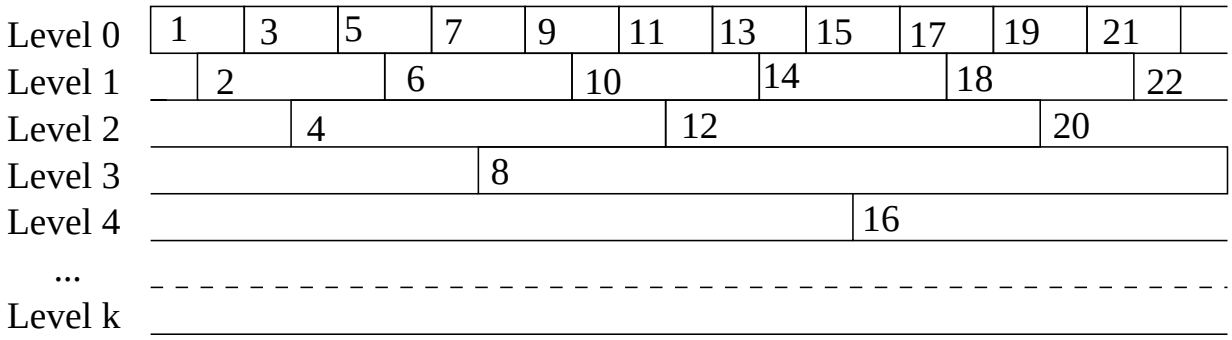


Figure 5.2: The Interleaved Stratified Timer Wheels Data Structure

as a fraction of the maximum line speed R of the router, as opposed to absolute rates that do not take the router's capabilities into consideration. The algorithm is able to achieve this performance using a group of timer wheels [48], stratified into levels such that the timer wheel at level i contains only flows with service rates between $\frac{R}{2^{i+1}}$ and $\frac{R}{2^i}$. Because of the different service rates, each timer wheel has its slots ordered to handle timeslots that are further apart than the previous level, as shown in Figure 5.1. The size of timer wheel elements in each level is set to be the length of time necessary to transmit λ bytes at the lowest service rate available at that level. For example, since the lowest service rate in level 1 is $\frac{R}{4}$, the length of a timer wheel element in level 1 is 4λ virtual time units. Packet arrival times are rounded down to the start of the slot, which makes insertions into the sorted container easier, reducing the complexity of the scheduler.

Among the levels, the timer wheels are interleaved to create a global ordering of the wheel elements, shown in Figure 5.2. By interleaving the levels, searching the container in order is reduced to a linear search.

The combination of stratification and interleaving creates a structure known as an *Interleaved Stratified Timer Wheel (ISTW)*. By having the higher service-rate levels of an ISTW use timeslots that cover smaller time windows, but are visited more often, the algorithm is able to give different rates of service to different flows. Each timeslot (henceforth referred to as a bucket) is essentially a queue of flows, so inserting a flow into a timeslot is equivalent to an enqueue operation, and removing a flow a dequeue operation.

The SI-WF^{2}Q scheduler uses two ISTWs, each of which has a different meaning for the timeslots. In the case of the first ISTW, each slot indicates the virtual time when a packet starts being transmitted, and the second ISTW orders packets according to when the router would finish transmitting the packet. It has been shown that sorting packets by either their start or finish time is insufficient to maintain certain fairness guarantees; however, sorting by both times can be used to achieve this property [3]. Sorting by both times allows the scheduler to avoid the problems with multiple large packets that have the same start time competing with a small

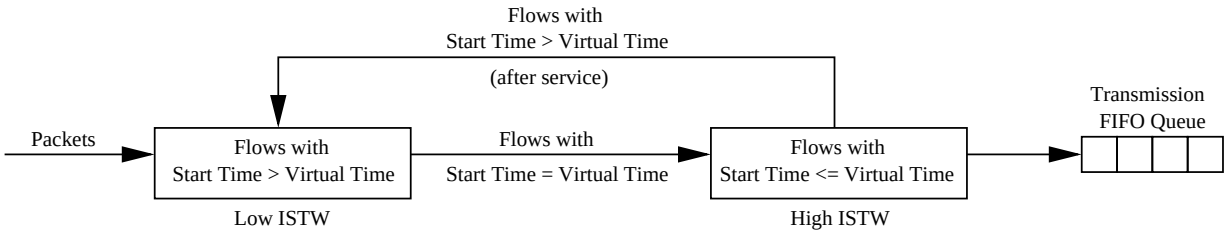


Figure 5.3: Simplified Scheduler Structure

packet with the same start time but a much higher service rate. Considering the finish time in this case allows the scheduler to transmit the small packet (with the earliest finish time) first so that it does not miss its transmission deadline. Therefore, two ISTW structures are used in this scheduler to handle sorting the packets by both their start and finish times. The first ISTW is called the *Low ISTW*, and sorts packets according to their start times. The second ISTW is referred to as the *High ISTW* and sorts packets according to their respective finish times.

Each ISTW structure is defined to have k levels. As mentioned above, for each level $i \leq k$, the maximum flow rate at level i is $\frac{R}{2^i}$, where R is the maximum line rate of the router. This means that for a scheduler implementation of 16 levels and a router with a maximum line rate of 1 Gbps, flows can have service rates between 1 Gbps and 15 Kbps.

There are 3 major parts to this scheduler, shown in Figure 5.3. First, flows are inserted into the Low ISTW. Next, an ISTW thread chooses the next timeslot with flows in it and transfers those flows to the High ISTW container. In the High container, the flows are sorted (this time based on the finish time of a packet) and the next eligible flow is put into a FIFO queue to be transmitted. The principle idea is that while a packet is being transmitted, the processor is not in use, so it can be used to perform lookup operations within the stratified timer wheels and keep the transmit queue filled.

5.2 Scheduler Operations

This section describes the basic operations that the SI-WF^{2}Q packet scheduler performs, in order to better understand the implementation.

5.2.1 Insertion

Inserting flows into the ISTW is quite simple, compared to inserting data into other sorted data structures, such as trees or priority queues. The start or finish time is rounded down to the nearest bucket time at the appropriate ISTW level. The flow is inserted bucket along with any

other flows that have the same rounded start time. Rounding the packet times allows the insertion operation to execute in constant time.

Searching the ISTWs

The most important operation that this scheduler performs is finding the next packet with the smallest timestamp (either start or finish time). Since the scheduler is modeled as a single-threaded algorithm, it is essential that this operation executes in a constant number of instructions each time. In this way, while a packet is being transmitted it can be guaranteed that another packet is found in a short enough time that the transmitter is kept fully utilized. If the operation has non-constant execution complexity, then as packet rates increase, so would the amount of time required to find the next packet to transmit.

The `find_next_flow` operation is facilitated by storing a bitmap that indicates which levels have flows stored in them. Finding the first bit set in that bitmap gives the lowest (i.e., highest service rate) level i in the ISTW with a non-empty bucket. Next, the current virtual time is rounded down to the nearest timeslot at level i to determine the first bucket to search. If that bucket is empty, the time is increased by 2^i and the bucket corresponding to that time is checked. The size of the time increment can be determined based on the first bit set in `anybits`, since there are no flows in the lower levels, and therefore no need to search them. This process is repeated until a non-empty bucket is found, and since the bitmap shows a non-empty bucket, the operation is shown in [28] to end within a constant amount of time. Finding the first bit set in a register is a non-constant task, however, the IXP assembly language provides the constant-time `ffs` instruction to perform this task. This instruction is essential in making the ISTW operations constant-time.

As an example, refer to Figure 5.2, with the next flow in bucket 20 and a virtual time of 6. The bitmap will show that level 2 has a non-empty bucket. Rounding the virtual time of 6 down to the nearest timeslot in level 2 results in the `find_next_flow` operation starting at bucket 4. After bucket 4, buckets 8, 12 and 16 (incrementing the searched bucket by increments of 2^2) are checked before the search finds bucket 20, which is non-empty. Also, if there is a flow in any of the earlier buckets, that is, those searched first, this search algorithm will discover it.

Transferring Flows

The `transfer_eligible` operation is used to move flows from one of the Low ISTW container to the High container or out of the High container for transmission. This operation is particularly important, as it must ensure fairness among flows based on their required service rate. Again, a bitmap `frontbits` is used to maintain which levels have flows waiting to be transferred to the High container. The lowest bit set in `frontbits` indicates the bucket to transfer a flow from and by using the bitmap, this operation always transfers the highest service rate flows first. Again, the `ffs` instruction is essential to searching `frontbits` and finding the highest-rate flow to transfer

in constant-time.

Iterating Through the Low ISTW

Whenever a packet is transferred to the FIFO queue for transmission, the virtual time is incremented by the size of the enqueued packet. At that point, a thread iterates through the Low ISTW, marking all packets with start times less than the new virtual time as being eligible for transmission. This operation takes time proportional to the size of the packet that was sent.

5.3 Scheduler Versions

5.3.1 Original Scheduler

The SI-WF^{2}Q scheduler described in [28] operates in the following manner. Packets are inserted according to their start time in the Low ISTW. The High ISTW requests flows from the Low container whenever High is empty. The Low ISTW sends the High container a flow using `transfer_eligible` or, if no packet is immediately available and the High container is empty, uses the `find_next_flow` to find the next flow to service.

There are several differences between the implementation of these methods and their description in [28] due to the multi-threaded nature of the implementation. First of all, since each ISTW method needs to access and modify data that is shared among threads (e.g., bucket and flow data), each method is a critical section that requires locking. Also, each ISTW can be implemented on a separate thread, so one thread can be transmitting packets while another thread is finding the next packet to transmit.

5.3.2 Packet-Only High ISTW

One problem with implementing the SI-WF^{2}Q scheduler as it is described in [28] is that flows are transferred from the Low to the High container and vice versa. Since the data transfer is bi-directional between the two ISTWs, this means that the next neighbor (NN) registers cannot be used for all data transfers. In this case, scratch buffer rings are the next fastest mechanism for transferring data between μEngines. However, using the scratch rings for this operation puts a large load on the data bus and scratch memory controller, which slows the scratch memory accesses on all of the μEngines. There is also the concern of priority and how to manage flows being inserted into an ISTW container from multiple sources, which would require more complex handling mechanisms and reduce the scheduler's packet processing rate.

To allow the scheduler to use only the NN registers between the μEngines, the data must all flow in one direction. When data goes from the High to Low container, it is because a flow does not have a head packet with a start time less than or equal to the current virtual time. In that case, the flow is transferred back to the Low ISTW to be scheduled for service at a later

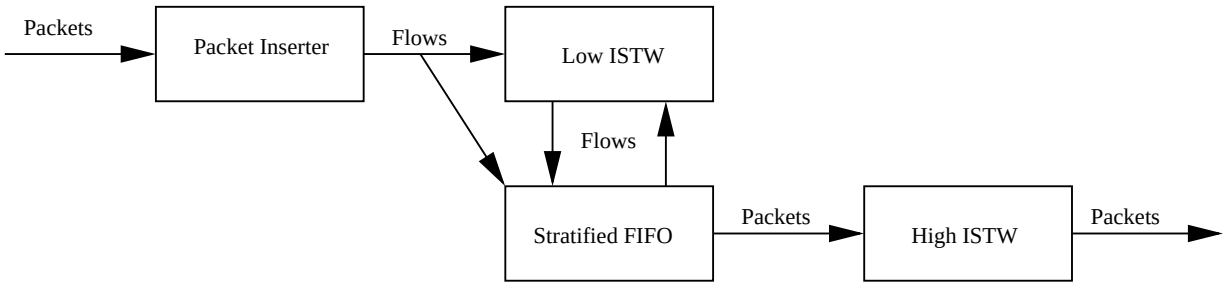


Figure 5.4: Structure of the Packet-Only High ISTW Scheduler

time. The simple solution is therefore not to pass flows to the High container, so that no flows are sent backwards. In this case, when transferring data to the High ISTW, a packet is dequeued from a flow and that packet is passed to the High container. The flow can then be immediately re-scheduled for service in the Low ISTW.

Transferring only packet data does not solve the problem of passing data directly to the High container, bypassing the Low ISTW, which can happen when a packet is enqueued onto a flow that has a finish time less than the current virtual time. In this case, the flow immediately goes to the High container since the start time is set to the current virtual time. Using only the next neighbor registers, it is not possible to bypass the Low container, and using some other transfer mechanism (such as a scratch buffer ring) recreates the problem of prioritizing data coming from multiple sources. Therefore, an intermediate structure called the *Stratified FIFO* is used to buffer data between the two ISTW containers.

Stratified FIFO

The Stratified FIFO structure is an improvement on the original scheduler. It is an array of size k, where each array element stores a queue of flows. The Low ISTW inserts flows into the Stratified FIFO, and packets are taken from those flows and transferred to the High container, as shown in Figure 5.4.

To use the NN registers to transfer data between the ISTW containers, all packets must go through the Low ISTW. When inserting packets, the packet start time is compared against the current virtual time, and in the case that the start time is earlier, the packet is appended to the Stratified FIFO element at the packet's assigned level. This puts the packet at or near to the front of the queue for earlier transfer to the High ISTW.

```
transfer_eligible()
    level = ffs(stratified_fifo_bitmap)
    q = stratified_fifo[level].dequeue()
    sendToHigh(q.dequeue())
    pkt = q.peek()

    timeslot = getTimeslot(pkt.startTime)
    if (getBucket(timeslot, level) == getBucket(curTime, level))
        stratified_fifo[level].enqueue(q)
    else
        insert_into_istw(q)
```

Figure 5.5: Pseudocode for Simplified `transfer_eligible` on Low ISTW

The `transfer_eligible` Operation Using the Stratified FIFO

With the Stratified FIFO structure, the `transfer_eligible` operation simply takes a flow out of the first non-empty array element. According to the algorithm in [28], after a flow is serviced in the High ISTW, it can be transferred back to the Low ISTW. In order to simplify the code, when `transfer_eligible` is run on the Low ISTW, a flow is no longer sent to the High container. Instead, a packet is dequeued from the flow and the packet is transferred. The next packet (if any) in the queue is checked and if its start time is also equal to the current virtual time, the flow is enqueued into the Stratified FIFO again (see Figure 5.5). Otherwise, it is inserted into a bucket in the Low ISTW based on the start time of the new queue head.

The changes to this operation creates a weighted priority-queue style mechanism for transferring data to the High ISTW, with the packets being sorted by their start time. Transferring packets to the High container does not change the properties of the scheduler though, as when the High container services the flow, according to the original algorithm, the flow is also re-inserted into the ISTW buckets. This change in the algorithm only serves to increase the simplicity of the High container, as the ISTW container no longer stores buckets with flows with packets, only buckets with packets. Also, it makes the Low container simpler, as it no longer has to handle packets coming from both the High container and the packet inserter μEngines. Finally, this change allows for the next neighbor registers to be used in all data transfers between the μEngines, increasing efficiency and saving the shared memory and data buses for other memory accesses.

The `find_next_flow` Operation Using the Stratified FIFO

The inclusion of the Stratified FIFO also changes the `find_next_flow` operation. Previously, this operation dequeued a single flow from an ISTW bucket and passed it to the High ISTW for transmission. Implementing the scheduler using the Stratified FIFO structure, `find_next_flow` now enqueues the entire bucket queue into the appropriate Stratified FIFO queue. On the IXP2400 network processor, this task is achieved by two enqueue operations, one standard enqueue operation using the front of the bucket queue, and a special `enqueue_tail` operation to adjust the tail pointer of the Stratified FIFO queue to point to the last element in the bucket queue. `enqueue_tail` is a special operation provided for the purpose of enqueuing a piece of data that uses multiple buffers, for example keeping a fragmented packet together in a queue. Data stored in a queue must always start with 4 bytes that are used by the queue as a pointer to the next element in the queue. Because of this pointer, splicing queues together requires two operations: enqueuing the first element of the second queue, and then updating the tail pointer to point to the last element in the second queue.

Updating the pointer in the last simply updating the tail pointer of the queue is sufficient to splice the queues together in a constant number of operations.

5.3.3 Low ISTW Only Scheduler

Another possible implementation of the scheduler is to exclude the High ISTW container from the scheduler. In this case, packets would be passed from the Stratified FIFO directly to the buffer ring that feeds the transmit μEngine. With the packet-only ISTW described in Section 5.3.2, the Stratified FIFO is an implementation detail used to reduce latency. Without the High ISTW, the Stratified FIFO replaces that container, transferring packets to the transmit μEngine in a stratified priority queueing scheme. This thesis describes an implementation of this version of the SI-WF^{2}Q scheduler.

Implemented in this manner, the scheduler no longer sorts the packets by both start and finish times and therefore small, high service rate packets may miss their finish times by a larger margin than the maximum packet size L. Packets are still served according to their service rates, but small packets are given the same priority as large packets with the same start time at the same level. This simplification translates into the larger error of $2^k L$ virtual time units shown in Lemma 6 of Section 6.2.

5.4 General Structure

This section outlines the general structure shared by all 3 versions of the scheduler, the Interleaved Stratified Timer Wheel and their contents. The timer wheels are implemented as arrays, each with 32 elements so that 32-bit bitmaps can properly describe the structures. These bitmaps

enable faster searches by identifying which levels have non-empty buckets so that empty buckets are not searched.

5.4.1 Queues

Most of the data structures in the SI-WF^{2}Q scheduler are queues. Every bucket (slot in a timer wheel) is essentially a queue of flows, and each flow contains a queue of packets. Also, the Stratified FIFO structure described in Section 5.3.2 is an array of queues of flows. This structure is important as the IXP has special memory controllers to handle queues in the SRAM (described in Section 4.3.2). The SRAM queue controller allows for atomic enqueue and dequeue operations, which is particularly useful when multiple threads are be removing flows from the same bucket. Another advantage of these queues is that since the operations are atomic, this means that only 1 memory access is required by the μEngines to enqueue or dequeue, as opposed to 2 accesses without the controller: one to get the respective head or tail element, and one to update the head or tail pointer. Before using a queue, the IXP2400 queue controller requires that a queue descriptor containing the head and tail pointers be read into the controller from SRAM. This adds an extra memory access, but is equivalent to reading the head or tail pointer from memory in any other queue implementation. Perhaps the most important reason for using the queue controller is that it handles any queue operation and all of the necessary error-case checking without using μEngine cycles. This optimization allows other threads in the μEngine to perform useful work while a thread waits for the SRAM queue controller to return a result.

In this implementation, each thread in the μEngine is assigned a small number of the 64 queue controller elements for its use. Each thread handling packet-insertion tasks is assigned one slot for loading flow queues in order to insert a packet into a flow. Threads executing ISTW operations have two queue slots assigned, one for the packet queue of a flow and one for either a bucket or Stratified FIFO queue. This structure allows Low ISTW threads to either dequeue a flow from the Stratified FIFO, or dequeue a packet from that flow and then re-enqueue the flow into the same Stratified FIFO queue, as can occur in `transfer_eligible`, all without having to reload the same queue into the controller. Because a queue is rarely loaded by the same thread twice in a row, threads always load and unload queue descriptors between uses, which ensures that queues are never loaded twice by different threads and that no packets are lost.

Freelists from the IXA portability library, shown in Figure 5.6, use queue elements as indices into an array; the address of the queue element acts as a pointer into an array of available blocks of memory. This implementation uses such a freelist for handling free memory for packet data. Flow data and packet data are stored as complete data structures with the first 4 bytes of each structure being the necessary pointer to the next element into the queue. Managing either the type of list shown in Figure 5.6 or in Figure 5.7 requires the same number of memory accesses, one to dequeue the data address and another access to get the data based on that address.

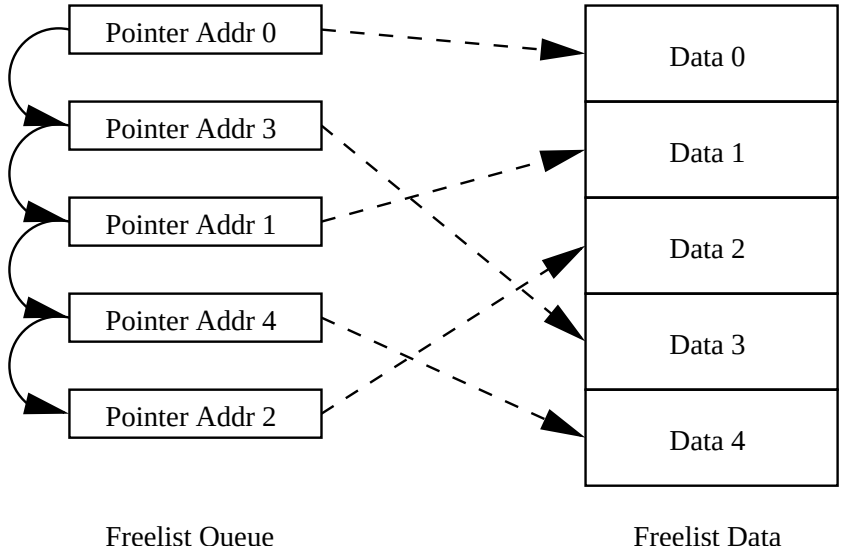


Figure 5.6: Freelist using Queues

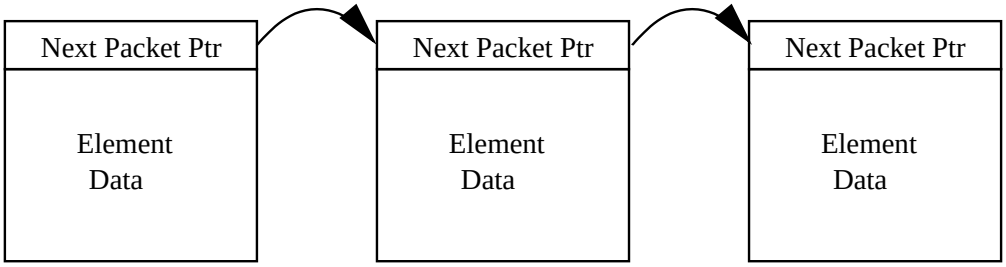


Figure 5.7: List of an Example Data Structure using Queue Pointers

5.4.2 Virtual Time

One of the most complex parts of emulating a GPS scheduler is handling the concept of *virtual time*. Virtual time must progress with actual amount of service (i.e., bytes sent). With the ISTW-based packet scheduler, virtual time is maintained by the ISTW containers, incremented by the High container each time a packet is sent to the transmission queue (i.e., whenever packets are transferred out of the High container). It is also incremented in the Low ISTW container to "jump" over empty buckets, this way, a bucket is only ever searched when the current virtual time is equal to or greater than the bucket's timeslot. The virtual time is shared among the threads and used by the scheduler to keep the ISTW containers synchronized. Because of the need for all the threads in the scheduler to access the current virtual time, it is stored in the scratch memory, the fastest memory accessible by all the μEngines.

In this design, the atomic scratch operation `add` is used to increment the virtual time, ensuring that any read access only accesses the current data. However, some flexibility is required here with regards to staleness, since it is quite possible that while a packet is being inserted into an ISTW the virtual time could change. Trying to ensure that such a case never happens would require synchronization among the two ISTWs, which would cause a significant performance degradation.

5.4.3 Memory

Other than the virtual time, the only data stored in the scratch memory is packet metadata, which is put into the scratch rings for transfer among μEngines. The data for each bucket in an ISTW is small enough that it could potentially be stored in scratch memory, but then the SRAM queue controller could not be used, as a bucket's queue descriptor must be stored in SRAM. Also, since it is possible to store additional data regarding the buckets in local memory, only the SRAM is necessary. If the ISTW container was shared by multiple μEngines for load balancing purposes, the scratch table would likely have to store data that normally is stored in local memory.

The raw packet data is stored in DRAM. DRAM has dedicated data buses to the media buffers and has the largest amount of space. Also, the raw packet data is rarely accessed, since most packet processing operations can be executed using the SRAM packet metadata. The packet data in DRAM is primarily accessed to read a packet's IP header for performing classification and route lookups. The scheduler itself does not access the DRAM.

5.5 Partitioning the Microblocks

Once the design of an application is complete, it is necessary to partition the computational code into microblocks. Figure 5.8 shows a general router. Note, most of the functionality concerns are processing the IP header and performing a lookup. Indeed, the Click Router Project [30], uses 20 microblocks (referred to as Click elements) in their implementation of a basic IP router.

Operation	SRAM	DRAM
Packet Receive	1	1
IP Processing	2	1
Route Lookup	5+	0
Classification	1-2	1
Deficit Round Robin Scheduler	1	0
FVC Scheduler	49	0
Packet Transmit	2	1

Table 5.1: Packet Processing Operations and Number of Memory Accesses Required

As shown in Table 5.1, IP header processing and routing lookups involve a large number of the memory accesses, since for large routing tables, 5 or more accesses may be required. The number of memory accesses is very important because they can quickly become the bottlenecks in packet processing. If each thread is constantly accessing memory, then the μEngine is going to be idle while its threads are waiting for the accesses to complete. Moreover, a large amount of memory accessing increases the delay before a packet is transmitted. The majority of the other accesses occur in the scheduler, as packets and flows are enqueued and dequeued. Classification of a packet based on a hash of the IP header may also take a substantial number of accesses, depending on the size of the hash table and the number of flows registered on the packet processor. For this reason, classification generally occurs during IP header processing, so as to save a DRAM memory access by reusing the header data.

Figure 5.8 contains a transmission queue manager and a scheduler. This structure represents a more general implementation, such as with Deficit Round Robin, where the scheduler can tell the queue manager from which queue to remove a packet. Since the SI-WF^{2}Q scheduler requires detailed knowledge of the packets within a queue, it performs both tasks. Thus, the scheduler and queue manager from Figure 5.8 can be replaced with the five microblocks shown in Figure 5.9.

In this design, the first microblock of the scheduler is responsible for inserting packets into the ISTW containers. In this case, if the packet's start time is less than the virtual time of the scheduler, the packet is moved into the High ISTW container; otherwise, it is inserted into the Low ISTW. In either container, the insertion microblock finds the appropriate flow and adds the packet to its queue. The ISTW containers compute the next flow to be removed from the container, and transfer it to the next microblock, either the High container's packet insertion microblock or the transmit microblock.

However, since there are only eight μEngines available for the set of packet processing microblocks shown in Figures 5.8 and 5.9, it is obviously impossible to assign each microblock to a separate μEngine. Only with 2 IXP processors, for a total of 16 μEngines, is it possible to assign

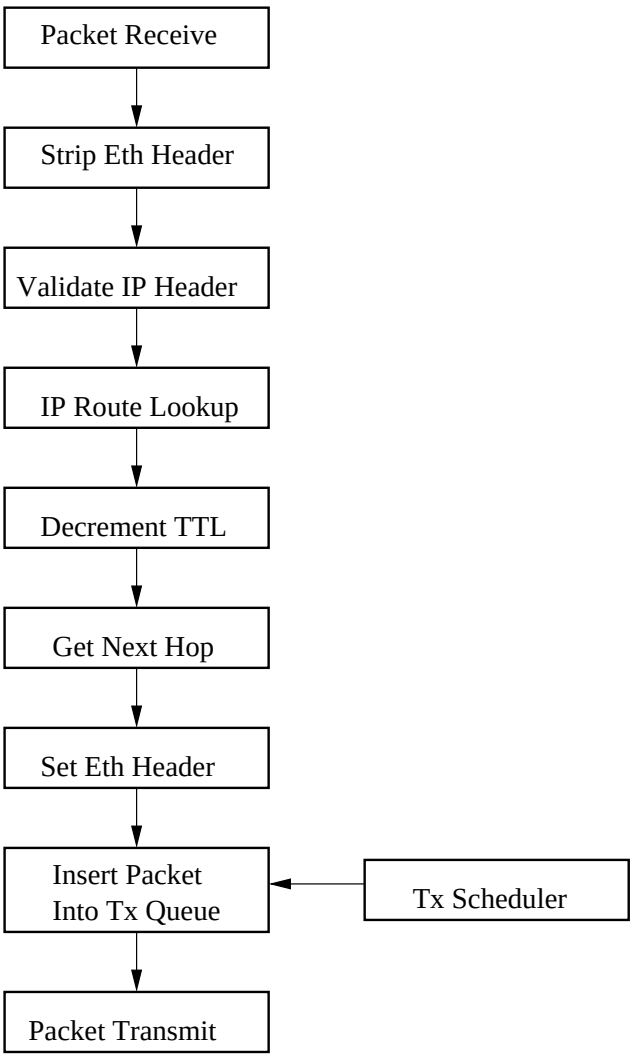


Figure 5.8: Microblocks Required for Packet Processing in a General Router

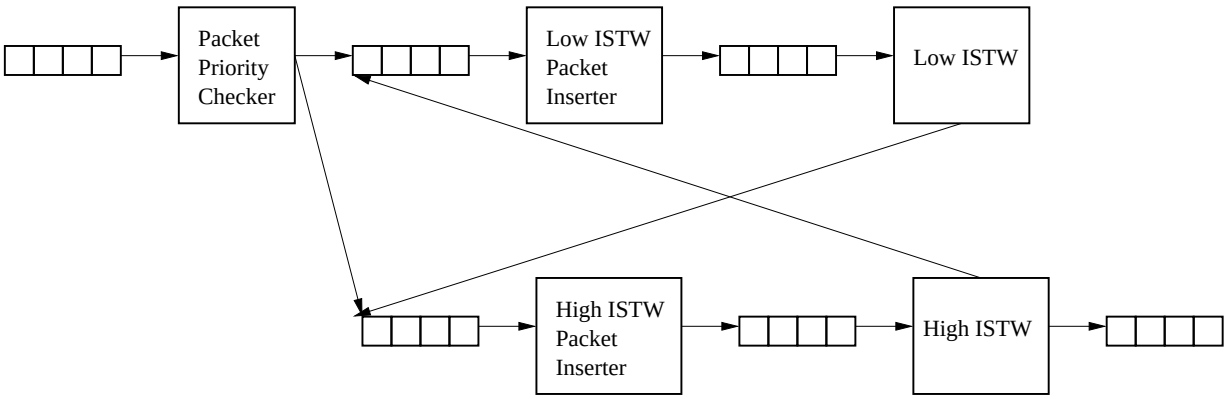


Figure 5.9: Microblocks for the Implementation of the FVC Scheduler

each scheduler microblock to its own μEngine. However, such an assignment would leave some of the μEngines performing redundant work. For example, each μEngine would be required to get the flow data from SRAM, often for very minor uses. By combining some of the microblocks, it is possible to improve the synchronization costs and latency due to redundant memory accesses. Therefore, the "High ISTW Packet Inserter" microblock is combined with the "High ISTW" microblock. With the inclusion of the Stratified FIFO data structure, it is also possible to combine the "Packet Priority Checker" and the "Low ISTW Packet Inserter" to insert a packet into its flow queue and insert the flow into either the Low ISTW or the Stratified FIFO. The actual microblock assignment is shown in Figure 5.10, which also assumes the dual IXP configuration. This configuration is implemented using an example IPv4 packet forwarder provided by Intel, so all of the microblock allocation is done by this software. The five microblocks of the scheduler are inserted onto 3 μEngines: one to handle insertion into the Low ISTW, one to perform the Low ISTW functions and one μEngine to handle all of the High ISTW functions. The Low container insertion code is assigned to its own μEngine because of the need to insert the packet into a flow before it is inserted into the ISTW container. The High container insertion code receives a flow, which requires significantly less processor time to handle.

In this design, threads in the first μEngine continuously loop through the packet insertion tasks, taking packets off of the scratch ring and passing them to the Low ISTW μEngine for insertion. Packets in the second μEngine, which handles the Low ISTW operations, continuously loop through the tasks required to remove flows from the ISTW and transmit packets from those flows. Data is passed from the first μEngine to the second using the next neighbor registers, which reduces the packet processing latency.

The Stratified FIFO structure and its data must be on the same μEngine as the Low ISTW. This structure simplifies the `find_next_flow` operation by allowing a constant number of operations to transfer all flows in a bucket to the Stratified FIFO, as opposed to looping to transfer

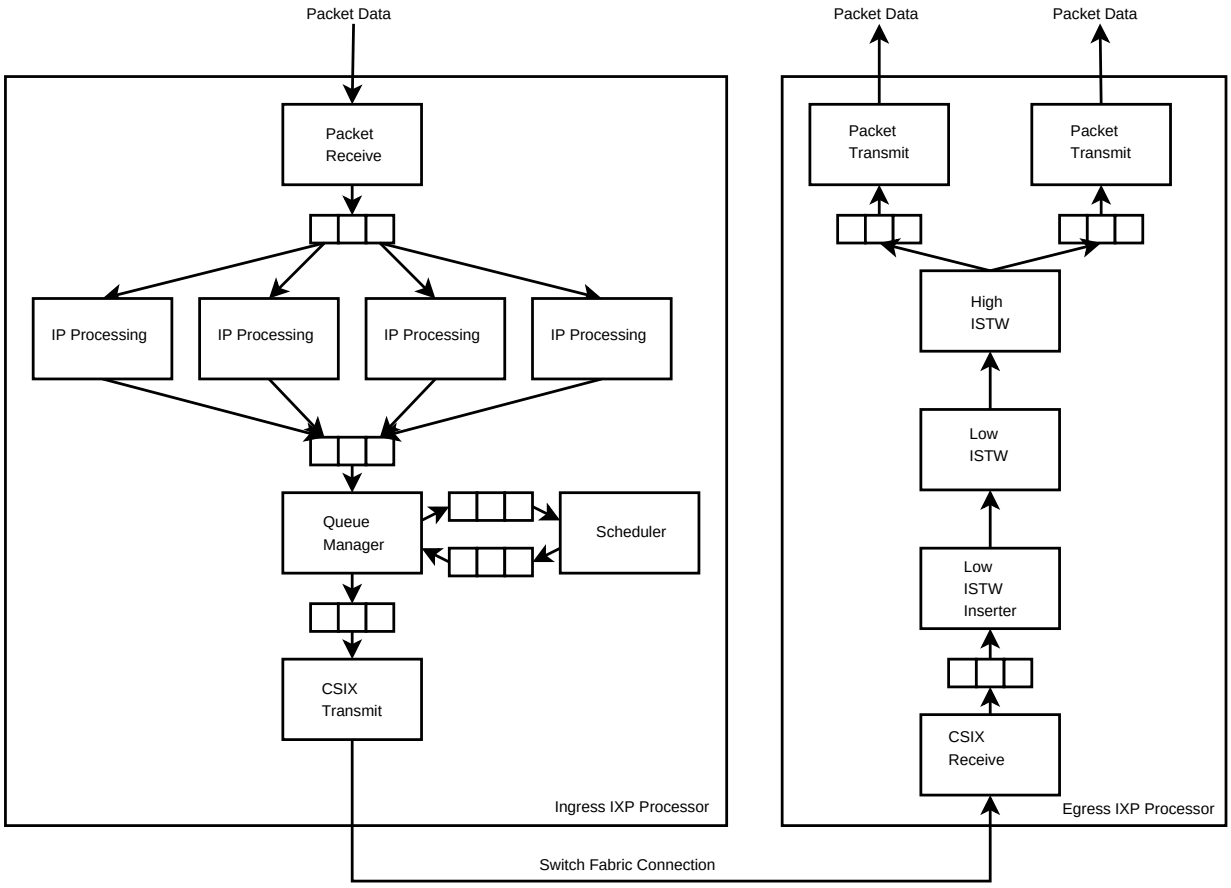


Figure 5.10: μEngine Layout of Packet Processor in Dual IXP Configuration

each flow individually. Also, during the `transfer_eligibile` operation, flows can be inserted back into the Low ISTW. Having the Low ISTW and Stratified FIFO structures in the same μEngine lowers the overhead of frequently passing flows between the two data structures.

It should be noted is that there is no longer a path that allows packets to travel immediately to the High container. This structure is done to create a pipeline formation within the scheduler, so the Next Neighbor registers can be used to transfer data, as they are the fastest means of doing so. The path directly to the High ISTW now goes through the Low container with a very small penalty, as explained in section 5.3.2.

Chapter 6

Scheduler Implementation

This chapter covers the implementation details of the Low ISTW SI-WF^{2}Q scheduler adaptation. Much of this implementation, particularly the implementation details regarding the ISTW container, can be reused for the versions of the scheduler described in Section 5.3.

6.1 Implementation of the Low-Only Scheduler

This section explains how the packets are sorted and otherwise manipulated as they pass through the scheduler. This thesis implements an adaption of the SI-WF^{2}Q scheduler with only the Low ISTW, so after a packet has been inserted into a flow, that flow is inserted into the Low ISTW, where it is sorted according to its start time (see Figure 6.1). Next, flows are transferred from the ISTW to the Stratified FIFO, and in that data structure, packets are removed from the flow and sent to the scratch buffer ring for the transmit μEngine.

6.1.1 Inserting Packets into Flows

The operations required to insert packets into flows are very simple, outlined in Figure 6.2. First, a μEngine thread gets a packet from a scratch ring, which have been put there by the μEngine that sets a packet's ethernet header. Since this implementation is only a prototype, classification is performed here to minimize the impact of the scheduler on the existing code base, which does not perform any classification functions. In this prototype, classification is performed by a separate set μEngine, and takes less execution than the scheduler, so it does not affect the SI-WF^{2}Q scheduler's performance. Classification steps involve loading the IP header of the packet into the DRAM transfer registers and classifying the packet based on its header. Classification is done by performing a hash on the IP header. This hash is used as the key into a hash table that contains all of the flows registered in the packet scheduler. Once the packet is matched with a flow, free SRAM memory is reserved for the packet metadata. The packet data, shown in Table

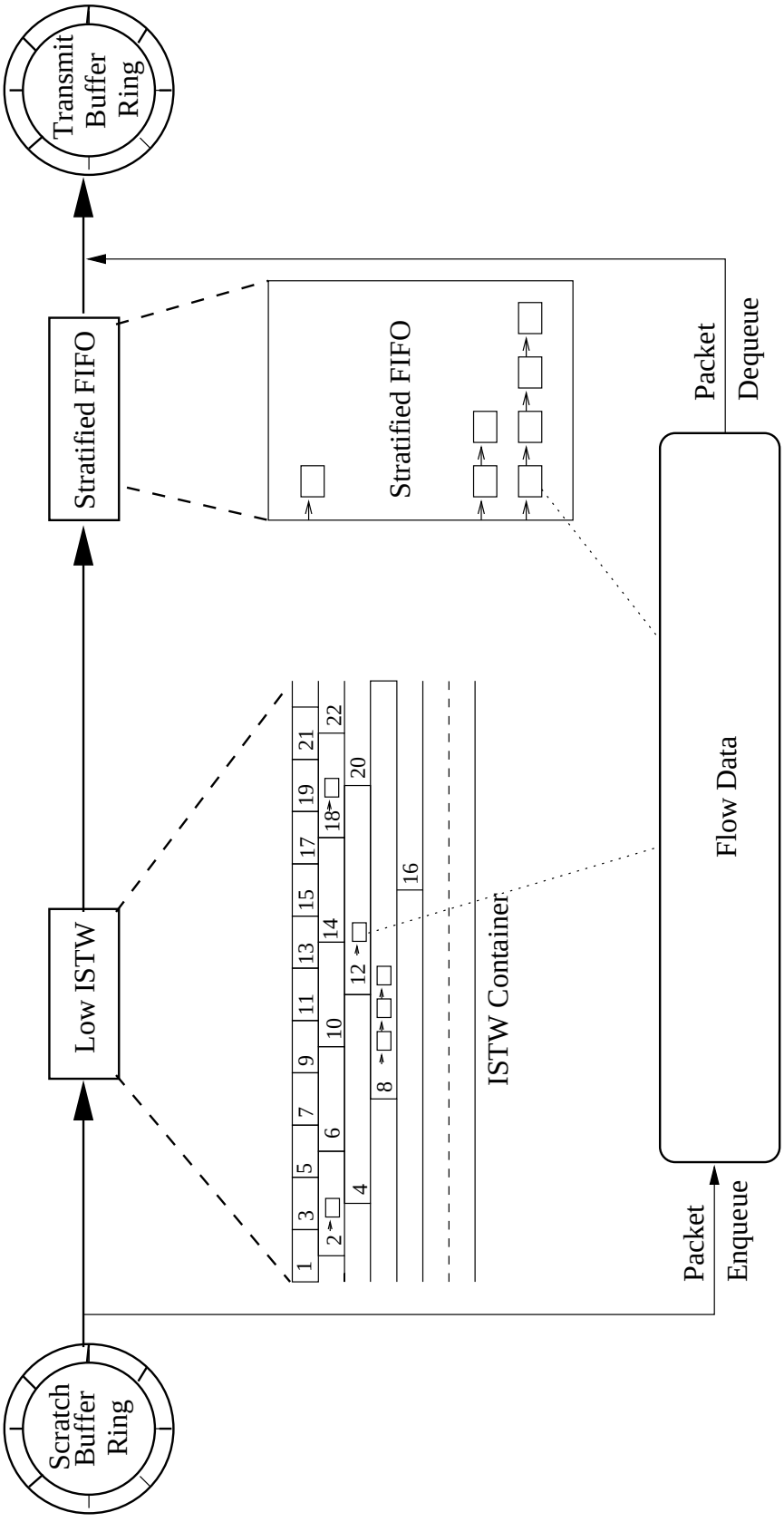


Figure 6.1: General Packet Flow Through the Scheduler

```
//read packet from scratch and get the IP header
sratch_packet_data = get_data_from_scratch_ring()

//classification steps
dram_addr = get_dram_packet_address(sratch_packet_data)
ip_hdr = read_ip_header_from_dram(dram_addr)
key = hash(ip_hdr)
flow = match_headers(key, ip_hdr)

//load packet into flow and update
sram_addr = get_free_sram()
plen = read_pkt_metadata(sratch_packet_data, LENGTH)
set_istw_packet_data()
load_queue(flow, thread_slot)
enqueue(thread_slot, sram_addr)
unload_queue(thread_slot, flow)

if (packet_is_queue_head)
    finish = calculate_new_finish_time()
    update_flow(finish)
    send_to_low_istw(flow)
```

Figure 6.2: Packet Insertion Pseudocode

6.1 contains the data from the scratch ring, as well as the packet length. The extra metadata, while already part of the dispatch loop metadata, is stored here to reduce the number of accesses required to get the information.

The SRAM address for this metadata (i.e., a pointer to the packet) is enqueued on the flow's packet queue and the queue status is checked. If the flow's queue is empty before the packet is inserted (i.e., the packet is the head of the queue), the flow's address is passed to the Low ISTW. In addition, the finish time for that flow (one of the pieces of metadata shown in Figure 6.2) is increased based on the packet length and the flow's rate. If the queue is not empty, then no further work is done, since these calculations are performed when the packet becomes the head of the queue.

6.1.2 Low ISTW

The Low ISTW has the same basic behavior as outlined in [28] and Section 5.2 of this thesis, using the Stratified FIFO structure described in Section 5.3.2. The general pseudocode in Figure 6.3

ISTW Packet Data
Next Packet Address
First DRAM buffer ID (scratch ring data)
Last DRAM buffer ID (scratch ring data)
Output port and class data (scratch ring data)
Packet start time

Table 6.1: Contents of the Packet Data Structure

ISTW Flow Data
Next Flow Address
Flow ISTW Level
Flow Rate
Flow Timing Data
Flow Lock (2 words)
Flow Queue Descriptor (3 words)

Table 6.2: Contents of the Flow Data Structure

```
[flow, start_time] = get_new_flow()
if (flow)
    if (start_time == cur_virt_time)
        insert_into_stratified_fifo(flow)
    else
        insert_into_istw(flow, start_time)

if (stratified_fifo.isEmpty())
    find_next_flow()

transfer_eligible()
```

Figure 6.3: Low ISTW Pseudocode

outlines these operations. Simply put, a thread executing the Low ISTW operations first checks for new flows from the operations described in the previous section. If there is a new flow, it is inserted into one of the ISTW bucket containers or the Stratified FIFO structure, depending on the start time of the packet. Both data structures are little more than sets of queues, so inserting a flow into either structure involves enqueuing the flow into the appropriate queue.

If the Stratified FIFO is empty, the thread executes the `find_next_flow` operation, which selects the flow with the earliest rounded start time from the Low ISTW and transfers it to the Stratified FIFO. Internally, this involves finding the next non-empty ISTW bucket and dequeuing the flows from the bucket and enqueuing them into the appropriate Stratified FIFO queue. The `transfer_elibigle` function takes flows from the Stratified FIFO and transmits them, which in this case involves dequeuing a packet from the flow and putting the packet onto a scratch memory buffer that is accessed by the μEngine responsible for transmitting packets. The flow is then re-inserted into either the Low ISTW or the Stratified FIFO, based on the rounded start time of the next packet in the flow.

ISTW Buckets

As mentioned previously, this implementation of the SI-WF^{2}Q scheduler uses 32 buckets in each level of the ISTW so that 32-bit integers can be used as bitmaps to describe the levels. The bitmaps are used as a local cache to identify which buckets contains flows. One integer bitmap is used per ISTW level, so that a 1 at position i signifies a non-empty bucket in slot i at that level. In addition to the per-level bitmap, there is also a bitmap `anybits` that indicates which levels contain flows. So if the bitmap for level j is non-zero, then bit j of `anybits` is set to 1. The `find_next_flow` operation only needs to check the bitmaps for a non-empty bucket, meaning that a thread only needs to make one memory access. This optimization is important to reduce the number of cycles a thread spends waiting for memory accesses to complete and reduce the total number of memory accesses required.

The `anybits` bitmap is stored in a GPR register local to the μEngine, while the level bitmaps are stored in local memory. This organization allows searches for non-empty buckets to be performed without requiring access to memory outside the μEngine, making such searches more efficient. The only data stored in SRAM are the queue descriptors for each bucket.

6.1.3 Virtual Time Implementation

As discussed earlier, the implementation in this thesis only uses the Low ISTW, and therefore virtual time is actually incremented in the Low ISTW threads, and read by those threads and packet insertion threads (which calculate the service times of new packets). In this case, it is actually possible to move the service time calculations to the Low ISTW threads to implement virtual time as a global register within the associated μEngine. This approach results in more work for a Low ISTW thread, but only when a packet is inserted into an empty flow, since timing calculations are done in `transfer_eligible` after service to calculate the service time of the next packet in the flow.

6.2 Service Guarantees

This section describes the mathematical proofs regarding the fairness and delay guarantees of a scheduler implemented using only the Low ISTW container. To make the analysis similar to [28], the Stratified FIFO structure is viewed as a substitute for the High ISTW container, as both store flows that are about to be serviced. In this case, the Stratified FIFO becomes a simple stratified priority queue scheduler. The lemmas here are based on the proofs in [28] and use the same terms as they are defined in that paper.

6.2.1 Terms and Definitions

R The maximum link speed of the router

r_i The relative rate of flow i

S The start time in virtual time units for a packet

F The finish time in virtual time units for a packet

λ The size of a time slot in an ISTW

L The maximum packet size

k_i The service level for flow i

l_i The length of the head packet for flow i

6.2.2 Service Rate Guarantees

Lemma 5: At any virtual time V, the following inequality holds for future virtual times V', where only flows with level $\leq x$ are being serviced in the time interval $[V, V']$.

$$\sum_{i:k_i \leq x \wedge S_i \leq V'} l_i + \sum_{i:k_i \leq x \wedge S_i \leq V'} (V' - F_i) r_i \leq V' - V$$

If some flow j does not have any packets in its queue, or is ineligible to transmit, $l_j = 0$ and $F_j \geq V$.

Proof. By induction over all possible events. The base case of an empty system is trivial.

Packet Enqueue: Packet p with size l_p arrives at the head of the queue, which increments $\sum l_i$ by l_p, and also decrements $\sum (V' - F_i) r_i$ by l_p. Therefore, the lemma holds.

Packet Transfer: No relevant variables are changed.

Virtual Time Jump: After a virtual time jump, $S_i \geq V$ is true for all flows, and Lemma 4 holds.

Packet Service: Serving a packet p with size l_p increments the current virtual time V by l_p. It also decreases $\sum l_i$ by l_p. The next packet in the flow's queue is handled by the *Packet Enqueue* case. Therefore, the lemma holds.

□

Corollary 1: At any virtual time V, the following inequality holds for future virtual times V' for the set of all flows with maximum level x, where only flows with level $\leq x$ are being serviced in the time interval $[V, V']$.

$$\sum_{i:k_i \leq x \wedge S_i \leq V'} (V' - S_i) r_i \leq V' - V$$

Proof. This corollary is a restatement of Lemma 5, changing $\sum(V'-F_i)r_i$ to $\sum(V'-S_i)r_i$. Now,

$$\begin{aligned}\sum_{i:k_i\le x\wedge S_i\le V'}(V'-F_i)r_i &= \sum_{i:k_i\le x\wedge S_i\le V'}(V'-(S_i+\frac{l_i}{r_i}))r_i\\ &= \sum_{i:k_i\le x\wedge S_i\le V'}(V'-S_i)r_i - \sum_{i:k_i\le x\wedge S_i\le V'} l_i\end{aligned}$$

Therefore,

$$\begin{aligned}&\sum_{i:k_i\le x\wedge S_i\le V'} l_i + \sum_{i:k_i\le x\wedge S_i\le V'}(V'-F_i)r_i\\ &= \sum_{i:k_i\le x\wedge S_i\le V'} l_i + \sum_{i:k_i\le x\wedge S_i\le V'}(V'-S_i)r_i - \sum_{i:k_i\le x\wedge S_i\le V'} l_i\\ &= \sum_{i:k_i\le x\wedge S_i\le V'}(V'-S_i)r_i \le V'-V\end{aligned}$$

□

Lemma 6: For any flow i with level k,

$$S_i \ge V - 2^k L$$

Proof. By contradiction.

The only time that could lead to a violation of this lemma is when flows with level $\le k$ are backlogged. Assume that at V_1 the lemma holds. A packet p with start time S_1 and length l_1 is served and afterwards, at V_2, there is a packet q with start time S_2, such that $S_2 + 2^{k_2}L < V_2$. There are 3 cases, based on the relationships between S_1, S_2, V_1 and V_2.

Case 1: Both packets p and q are eligible at V_1, that is, $S_1, S_2 \le V$. Since p is served before q, $k_1 \le k_2$. Applying Corollary 1 with $V = V_1$, $V' = S_2 + 2^{k_2}L$ and $x = k_2$ yields the following:

$$\sum_{i:k_i\le k_2\wedge S_i\le S_2+2^{k_2}L}(S_2+2^{k_2}L-S_i)r_i \le S_2+2^{k_2}L-V_1$$

Since $(S_2 + 2^{k_2}L - S_i)r_i \ge 0$ for both flows ,

$$\begin{aligned}(S_2+2^{k_2}L-S_2)r_2 &\le S_2+2^{k_2}L-V_1\\ (2^{k_2}L)r_2 &\le S_2+2^{k_2}L-V_1\\ \text{Since } r_2 \le \frac{1}{2^{k_2}},\ l_p \le L &\le S_2+2^{k_2}L-V_1\\ \text{Since } V_2-V_1 = l_p,\ V_2-V_1 &\le L \le S_2+2^{k_2}L-V_1\\ V_2 &\le S_2+2^{k_2}L\end{aligned}$$

Case 2: A virtual time jump happens before servicing q:

$$V_2 \leq S_2$$

Case 3: Packet q becomes eligible between V_1 and V_2. In that case,

$$S_2 \geq V_1 \geq V_2 - L$$
$$S_2 + L \geq V_2$$

□

Lemma 6 determines the maximum effective finish time, $\dot{F}$, for a packet. For the original SI-WF^{2}Q scheduler, $\dot{F} \leq S + \frac{l}{r} + 2^k\lambda$. For this adaptation of the scheduler, $\dot{F} \leq S + 2^k L \leq S + \frac{2L}{r}$.

6.3 Synchronization

In the original single-threaded algorithm in [28], there is no need to manage access to any of the data. However, with the 16 possible threads between the two μEngines used in this implementation, there will be access conflicts trying to lock high rate flows (those in the first couple of levels of the ISTW structure) due to the high rate of packets being enqueued onto and dequeued from the flow.

In most of the SI-WF^{2}Q data structures, the part of the data that requires synchronization is the queue data. While the SRAM controller allows for atomic enqueues and dequeues, the queue data has to be read from memory into the controller first before the queue can be used. In this case, it is necessary to guarantee that the queue descriptor is not read into hardware twice, otherwise when the queue data is written back into memory, some of the enqueued data is lost or overwritten.

6.3.1 Classifying flows

While not strictly associated with the SI-WF^{2}Q scheduler, classifying flows is the only action in this prototype that requires no synchronization. Since the classifier only compares, which is a read-only operation, the packet data to static flow data (the source and destination information), there is no need to guard access to this data. The only exception is when removing a flow from the scheduler's hash table. In this case, it is a matter of blocking all access to the flow hash-table when the flow is being removed. Once the flow queue is empty (or before if this is the desired action), the flow (and any packets still in the flow queue) can be removed from memory. Handling reservation requests for new flows could complicate removing flows, but is beyond the scope of this thesis.

6.3.2 Accessing Flow Data

Managing access to flows is very difficult due to the frequency of access. Every time a flow is inserted into a bucket, removed from a bucket and inserted into the Stratified FIFO, or removed from the Stratified FIFO, the `next_flow` pointer data in the flow changes. It is important this pointer is not overwritten or lost, as this would cause those flows already in the list to be lost and never be serviced. Therefore, inserts and removals of the flow from queues must be protected against multiple threads performing actions simultaneously. However, this is the responsibility of the queue that contains the flow, as the enqueue operation changes this data, making it necessary to lock either the buckets or the Stratified FIFO before enqueuing a flow to prevent the next flow pointer from being overwritten.

Every packet added or removed from the flow queue requires atomic access, which is trivial using the atomic enqueue and dequeue operations provided by the SRAM queue controller. However, reading the queue data from memory into the controller requires that the queue not already be loaded into the controller so that no packets are lost. As with moving flows between lists, access to the queues of packets within a single flow must also be protected so that no two threads can be performing this action at the same time.

To ensure that the `next_flow` pointer is not overwritten, a lock is associated with each flow. This lock is commonly referred to as a *ticket counter* lock, as it is analogous to each thread receiving a numbered ticket from a ticket dispenser (the first word) and waiting until the "currently serving ticket" value (the second word) matches their ticket. When a thread wishes to access a flow's data, it performs a `test_and_incr` atomically on the first word. This atomic operation returns the previous value of the word to the thread and increments the value of the ticket dispenser. The thread compares this value to the serving ticket (the second word), and when they match, the thread is allowed to access the flow data. Once the thread is finished, it increments the currently serving ticket value, allowing the next thread waiting to access the data. One drawback with this lock is that it requires polling SRAM to check the "currently serving" word. Also, since the lock is in SRAM, even if there are currently no threads accessing the memory, one SRAM access is still required to acquire the lock. Therefore, there is a long latency (at least 90 cycles) between requesting the lock and actually acquiring it.

Unfortunately, threads on two μEngines, the Packet Insertion μEngine and the ISTW μEngine containing the flow, could require access to a flow's data at any time. Therefore, it is conceivable that a total of 16 threads could all be waiting to access a flow's data, which would result in very slow progress for both μEngines.

6.3.3 Accessing the ISTW Buckets

The buckets or slots that make up the Interleaved Stratified Timer Wheels are accessed by two operations: `insert_into_istw` (from Figure 6.3) and `find_next_flow`. While inserting a flow into an ISTW only accesses a single bucket, the search performed in the `find_next_flow` operation

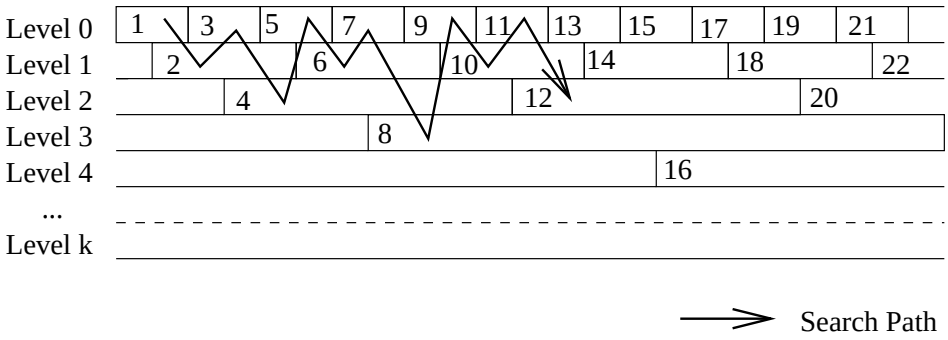


Figure 6.4: An Example Search Path for the `find_next_flow` Operation

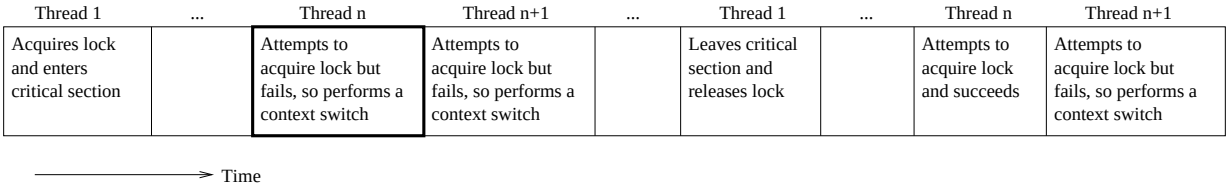
accesses multiple buckets on multiple levels, as shown in Figure 6.4. In either operation, bucket lists are accessed, as are the bucket bitmaps that store which buckets currently contain data.

If each bucket had its own lock, this would require that a thread trying to access the ISTW (for example in a `find_next_flow` search) acquire a large number of locks during an operation that must complete in as small a number of operations as possible. As such, a single lock is used to guard the entire ISTW structure. However, due to inclusion of the Stratified FIFO and the changes in the `find_next_flow` operation, the operation performs more useful work each time it is executed, since an entire bucket is emptied.

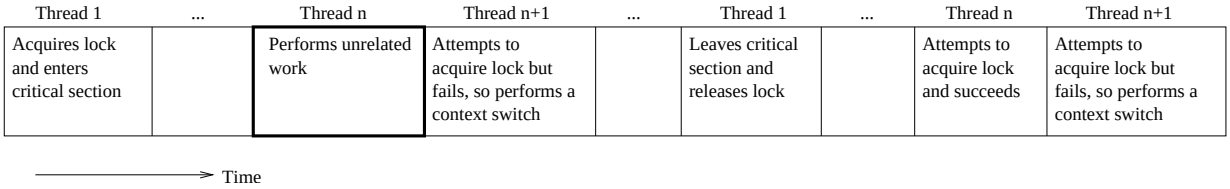
In this design, the ISTW buckets are only accessed by the μEngine threads that implement the ISTW, so locking the buckets is done using general purpose registers that are accessible to all of the μEngine threads. The `find_next_flow` operation has no flow data associated with the threads, so starvation, which would cause packets to miss their service window, is not a concern. However, inserting a flow into a bucket is a time sensitive operation, as the flow's packets may not be served in time if starvation occurs. However, it turns out that starvation is impossible with a local register-based lock. This is due to the round-robin style ordering of threads. For example, see Figure 6.5(a). If Thread 1 currently has an intra-μEngine, local register lock, and Threads n and $n+1$ are waiting on the lock, it is impossible for Thread $n+1$ to get the lock before Thread n. This is because once Thread 1 releases the lock, the μEngine thread arbiter guarantees that Thread n is executed before Thread $n+1$. As shown in Figure 6.5(b), it is possible for threads to acquire a lock out of order (ordering according to time spent waiting), but this is not a problem with accessing the buckets.

6.3.4 Accessing the Stratified FIFO

The Stratified FIFO is accessed by three operations: `insert_into_stratified_fifo` (from Figure 6.3), `find_next_flow` and `transfer_eligibile`. All of these operations enqueue and/or dequeue



(a) Thread Starvation is Impossible



(b) Acquiring Locks Out of Order

Figure 6.5: Arbitration and Synchronization between Threads

flows from a Stratified FIFO slot as well as adjust the bitmap indicating which slots contain flows to be transferred.

As the Stratified FIFO is local to a single μEngine, the locking mechanism is the same as the ISTW buckets, a single register local to the μEngine is used to guard access to all of the Stratified FIFO data.

6.3.5 Lock Hierarchy

To avoid deadlock among threads, the locks for each data structure are arranged in a hierarchy, are so that each data structure must be locked in a specific order. The hierarchy is as follows:

1. ISTW Bucket
2. Stratified FIFO
3. Flow Data

The order is defined as such because the ISTW buckets and Stratified FIFO slots contain the flows, so in order to determine which flow needs to be locked, it is necessary to dequeue the flow from the bucket or Stratified FIFO slot first.

6.4 Data Parallelism

Scalability is a concern with any packet processing application. As network speeds increase, it often becomes necessary to distribute the workload over several processors. This application already utilizes distributed processing, with each ISTW being managed by a single μEngine. Unfortunately, it appears that it may not be possible to further distribute the workload for this scheduler. This is primarily due to the configuration of the buckets in the ISTW. As shown in Figure 6.4, the `find_next_flow` operation could potentially traverse every level of the ISTW container. The search path means that it would be extremely difficult to partition the data structure for parallel access or processing without another container managing the flows being output by the partitioned containers. This new container would then likely become the bottleneck just as the implemented ISTW container is now.

6.5 Code Analysis

This section compares the implementation of the Low-Only SI-WF^{2}Q scheduler to Intel's example implementation of a Deficit Round Robin (DRR) scheduler. Like SI-WF^{2}Q, DRR [43] is a rate-based start-time scheduler that processes packets in constant time, however DRR does not provide constant-time delay properties. Both implementations use two μEngines on the IXP2400 network processor; DRR uses one μEngine to implement a queue manager and one to execute the DRR algorithm. DRR is a much simpler algorithm than SI-WF^{2}Q and does not use large data structures in shared memory, however, comparing the two implementations gives useful indications of processing effort and complexity, and also allows for comparison between the synchronization mechanisms used.

Table 6.3 shows the number of instructions and memory accesses for each processing stage. The table also shows the number of instructions and memory accesses that are used to synchronize threads in the scheduler. In this analysis, synchronization instructions include signalling other threads, acquiring locks and managing queues. Note that these numbers represent a minimum value, as the number of instructions and memory accesses increase if a thread is unable to acquire a lock on the first attempt.

As shown in the table, synchronization tasks represent 9% of the SI-WF^{2}Q code, and 5% of the DRR code. 47% of the SI-WF^{2}Q memory accesses are for synchronization purposes, compared to 31% of DRR memory accesses. The differences in synchronization overhead are primarily due to the number of concurrent threads in the DRR scheduler and how shared memory is used. The μEngine implementing the DRR algorithm uses only 2 threads: one to run the algorithm and one to handle messages from the queue manager. These two threads have little interaction and only local memory is used, so there is very little synchronization required. On the other hand, the μEngines in the SI-WF^{2}Q implementation are both sharing memory and there is a significant amount of interaction; for example, 9 of the 23 synchronization memory accesses are locking and

Table 6.3: Number of Instructions and Memory Accesses per Scheduler Operation

	SI-WF^{2}Q		Deficit Round Robin	
Task	Instructions	Memory Accesses	Instructions	Memory Accesses
Storing Packet	233	25	59	5
Marking Next Packet for Tx	154	7	55	1
Sending Packet	196	17	211	7
Total	583	49	325	13
Synchronization	51	23	16	4

unlocking flows.

6.6 Optimizations

After the initial development stage, the next step is to optimize the code in order to improve performance. A simple visual examination of the code shows that the majority of the performance penalties are related to the concurrency mechanisms, in particular locking the flow data and accessing the Stratified FIFO. This is because flow data and the Stratified FIFO are used in multiple operations, and locking the data prevents more than one thread from accessing it. This section describes one optimization that was implemented by making better use of the available hardware and removing some of the necessary locking mechanisms.

6.6.1 Maximal Use of the SRAM Queue Controller

One solution for improving performance is to increase the usage of the SRAM queue controller, which can hold 64 entries. With the initial development, when using a queue, a thread must first read the queue descriptor from memory into the controller, perform a set of dequeue and/or enqueue instructions, and then write the data from the controller back into memory so that another thread could use the data. This process is redundant, especially in the case of the Stratified FIFO queues, which are accessed every time `transfer_eligible` is executed.

It is often the case that frequently accessed queues, such as memory freelists, are not removed from the SRAM controller. Since the queues only take up a single slot, this is considered an acceptable tradeoff for not having to regularly load and unload the freelist queue data. Similarly, in this implementation, the Stratified FIFO is relatively small, having only 16 queues. 8 entries in the queue controller are already assigned to the Low ISTW threads for storing the Stratified FIFO queues, so if there is unassigned space for 8 more entries, it would be possible to leave the entire set of Stratified FIFO queues in the queue controller. Just like keeping freelists in the SRAM controller, keeping the Stratified FIFO queues in the queue controller improves performance in several ways:

1. All Stratified FIFO queues could be written to the SRAM queue controller during initialization, resulting in two fewer SRAM commands per queue access, since the queue data is no longer read to or written to the controller. Not having to manage these accesses means less processing and fewer context switches for the threads.

2. Fewer SRAM accesses also results in lower utilization of the SRAM data bus and memory controller, which results in faster SRAM accesses over all the μEngines.

3. It would no longer be necessary to lock the Stratified FIFO before accessing it, since only the Low ISTW μEngine accesses it. Since only a single thread can access any of the related bitmaps at a time and all queue operations are atomic; therefore, there is no chance of one thread overwriting another thread's changes.

4. Since locking the data structure is no longer required, it would be possible for one thread to be dequeuing flows in `transfer_eligible` while another is enqueuing flows. This is a significant improvement that would greatly increase the concurrent execution within the μEngine, since both operations currently require locking the Stratified FIFO, meaning that only one of the two operations could be executed at a time.

Table 6.4 shows how this optimization affects the implementation, in particular the `find_next_flow` and `transfer_eligible` operations. The total number of memory accesses is reduced by 6, and now synchronization accesses represent 40% of the total number of accesses. By reducing the number of memory accesses, this optimization reduces the number of cycles spent waiting for memory accesses to complete by at least 12%, which significantly reduces the total latency experienced by a packet.

Table 6.4: Number of Instructions and Memory Accesses for Optimized Scheduler

	SI-WF^{2}Q	
Task	Instructions	Memory Accesses
Marking Next Packet for Tx	144	4
Sending Packet	186	14
Total	563	43
Synchronization	45	17

Unfortunately, it is not possible to keep the bucket queues or the flow's packet queues in the SRAM controller due to the large number of them. Furthermore, keeping bucket queues in the queue controller would not provide any benefit, since each bucket is always emptied into the Stratified FIFO, leaving an empty queue that will not be used again until the timer wheel has looped around again.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

This thesis describes the implementation of a new weighted scheduling algorithm with constant fairness and delay characteristics on the Intel IXP2400 network processor. In the scheduler, flows are stratified by their relative service requirements and packet transmission is regulated by two sorted containers, called Interleaved Stratified Timer Wheels. By rounding the service times of the packets, the complexity of the sorted ISTW containers is reduced, and sorting the packets first by their start time and then by their finish time allows for favorable fairness and delay properties. On a multi-threaded, multi-engine network processor like the IXP2400, the containers can be separated into different engines to maximize concurrent execution.

Programming for network processors requires an in-depth understanding of the hardware's idiosyncracies. A programmer must balance the complexity and performance rates of the packet processing tasks with the number of processing engines available while still trying to take advantage inherent parallelism of the processing tasks. The sizes and speeds of the different levels of the memory hierarchy must be considered so as not to overflow the memory buses, and special hardware blocks should be used whenever possible to maximize performance. All of these objectives make it very difficult to create a packet processor with maximized performance. However, with an NPU's specialized programming language and a reasonable understanding of the NPU hardware, it is relatively simple to implement a packet processor with good performance. Further optimizations require an excellent background in concurrency and an extensive understanding of the network processor technology.

Two new modifications of the SI-WF^{2}Q scheduler are also proposed in this thesis. The first version transfers packets, rather than flows, from the Low ISTW to the High ISTW in order to reduce the processing latencies and complexity. This scheduler maintains the fairness and delay properties of the original scheduler. The second version uses only one of the ISTW containers, resulting in a stratified priority queue style of packet service. This version sacrifices

the constant scheduling properties while maintaining constant execution rates regardless of the rate of incoming packets at the router. Nevertheless, this version of the scheduler is able to maintain delay properties that are only dependant on the relative service rate of the flows.

7.2 Experiences

After the initial learning curve, it is relatively easy to program a packet processor for the IXP2400 network processor using the provided assembly language. First of all, the language provides some basic control structures such as loops and if-statements. Also, the macros that are inserted into a program at compile time allows for code reuse, especially when a set of them are combined into a library such as the IXA Portability Framework. Finally, the assembly language instructions, particularly the atomic shared memory operations, give the programmer the tools necessary to handle multiple threads and manage shared memory.

However, managing shared access is very complex and can require a disproportionate number of cycles, especially with a packet processor where the number of cycles used per packet must be minimized. Indeed, when implementing a filter to adjust the quality of video streams, the authors of [52] found that performance decreased due to the extra synchronization necessary when one μEngine processed packets with the StrongARM core processor; using all 6 μEngines and the core processor resulted in twice the performance of the core processor alone. These results, as well as the static analysis performed in Section 6.5, suggests that when modifying shared memory, handling concurrent access greatly increases the overhead. Moreover, modifying shared memory generally requires acquiring a lock in shared memory, which can increase the total processing latency of the packet.

Based on our experience implementing the Low-Only SI-WF^{2}Q scheduler, we speculate that many of the performance penalties are mitigated somewhat by large volumes of work. For example, if all of the threads on a μEngine are locking different flows, there is little or no penalty. Even if a small portion of the threads are trying to lock the same flow, the performance penalty of a small number of threads having to make additional attempts to lock the flows is offset by the other threads doing useful work during these failed attempts. In this example, the use of fine-grained locks, one lock per flow instance, is key to maintaining high performance. Additionally, pipelining an operation into multiple sequential steps can increase the number of threads that are able to perform useful work, further masking any delays due to synchronization. For example, if many threads are waiting for access to the Low ISTW, the performance penalties can be masked to a large degree if other threads are able to do useful work by servicing flows in the Stratified FIFO. Both of these examples follow the key design principle of the IXP2400 network processor, which was to take advantage of the inherent concurrency available when processing packets independently of other packets.

While it is possible to achieve high performance rates on the network processor by carefully utilizing the multiple threads and processors, doing so takes a significant effort from both the

designer and programmer. Requiring that threads acquire many locks increases the possibility of deadlock and requires more design and/or error checks to avoid these failures, as well as requiring processor cycles to actually acquire the locks. Also, some of the synchronization mechanisms, such as SRAM queues, on the IXP2400 can be quite cumbersome to use, requiring several cycles to configure usage instructions for access. These extra cycles cannot be masked by other threads, and therefore degrade performance, especially if the mechanisms are used heavily.

7.3 Future Work

7.3.1 Caching Flows

Keeping flow queues in the SRAM queue controller is another optimization to lower the number of memory accesses required by the scheduler. Since there are too many flows to keep all of them in the queue controller, the μEngine CAM unit could be used as a cache to store which flows queues are currently in the controller, using the memory address of the flow as a unique identifier. A cache hit would return an index into the queue controller, while a cache miss would require unloading a queue controller element and loading the desired flow queue descriptor. The CAM unit could also be used to hold a lock on the flow data, so that threads do not have to poll SRAM memory, which introduces a large delay.

However, the number of flows can increase exponentially at each level, making cache thrashing a distinct possibility. For example, a flow in level 2 of the ISTW could be serviced, and therefore in the cache. Subsequently, 16 flows in level 10 could be serviced, which would wipe out all previous data in the cache, since there are only 16 elements. If these flows have a much smaller service rate, they are not serviced again for a long period of time, making the act of caching them wasteful. Therefore, a simple LRU replacement scheme for caching data would be insufficient.

A caching scheme that would avoid thrashing is to only allow high-rate flows to be cached. By only allowing the lower levels to cache, it would ensure that slower flows are not pushing out the higher service rate flows. Since the higher service rate flows are accessed more frequently, they derive the most benefit from the cache. Lower rate flows are not processed any faster, but are processed with less delay since the higher rate flows are processed faster.

7.3.2 Implementing the Packet-Only High ISTW

The version of the SI-WF^{2}Q scheduler implemented in this thesis only uses the low ISTW. In Section 5.3.2, it was suggested that the SI-WF^{2}Q scheduler could be simplified by only transferring eligible packets to the high ISTW. This modified scheduler would maintain the constant fairness and delay properties of the original scheduler, while reducing the latency in transferring data among the containers. However, implementing the high container poses many challenges, since it requires a higher level of synchronization between the two ISTW containers. In particular, virtual time must now be stored in scratch memory so that it can be accessed by the two μEngines

executing the ISTW operations. Stale virtual time data then becomes a concern, as it is used as a starting point for `find_next_flow`.

7.3.3 Future of Network Processors

One obstacle network processors currently face is the lack of a high-level language capable of providing high packet processing performance. Such a language needs to allow a programmer to define processing tasks and automatically distribute these tasks to the packet processing engines of the NPU. In addition, the language needs to efficiently manage concurrent processing and memory accesses with minimal programmer input. Finally, this high level language should be able to use the specialized hardware blocks on a network processor to improve performance. A high-level language with these characteristics would allow programmers to quickly create applications for network processors that are able to process packets at the rates required by current data networks.

In order to define such a high-level language, a consensus must be reached regarding the primary tasks that a network processor is meant to perform. In this way, network processor manufacturers can ensure their products are able to efficiently execute these tasks, while still allowing for specializations beyond the primary requirements. At a minimum, this standardization would create an interface that a higher-level language can be built upon. Task partitioning and synchronization among tasks is much more difficult for network processors, and it still very difficult to achieve on more general-purpose hardware. To achieve this on network processors would require a profile or some description of the hardware which is used for task allocation. Again, this would require a standard for how the network processor is described and would likely require an agreement among NPU designers on a base architecture.

In a research environment, the flexibility of network processors is extremely useful. In particular, most network processor engines are run-time configurable and can have new functionality loaded onto them with minimal downtime of the processing engines. This ability makes network processors an attractive platform for active networking, which allows the users of the network to insert customized code onto the network nodes. This code is then executed on the packets as they pass through the nodes. In addition, with their ability to handle higher packet rates than general purpose processors, network processors could become the preferred means of testing new packet processing algorithms, such as the scheduling algorithms described in this thesis.

The flexibility of network processors is both a major advantage and a major disadvantage. On one hand, NPUs represent a cheap and efficient means of processing packets in almost any way that could be imagined. However, this flexibility also results in devices that can be extremely difficult to program at the desired performance rates. Making network processors easy to program is the most important next step, academically and commercially.

Bibliography

[1] Microsoft directx homepage. Website.

[2] Smita Bakshi and Daniel D. Gajski. Partitioning and pipelining for performance-constrained hardware/software systems. *IEEE Transaction on VLSI Systems, Dec. 1999.*, 1999.

[3] J. Bennett and H. Zhang. Wf^2q: Worst-case fair weighted fair queuing. *IEEE Infocom*, 1996.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] Ioannis Charitakis, Dionisios Pnevmatikatos, Evangelos Markatos, and Kostas Anagnostakis. Code generation for packet header intrusion analysis on the ixp1200 network processor. *Lecture Notes in Computer Science*, 2826, 2003.

[6] Douglas E. Comer. *Network Systems Design using Network Processors (Intel IXP2xxx Version)*. Pearson Education, Inc., 2006.

[7] Intel Corporation. Ixp2400 network processor. Technical report, 2004.

[8] Nvidia Corporation. *GPU Gems 2*. Addison/Wesley, 2005.

[9] Geoff Coulson, Gordon Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antonio Gomes, and Yimin Ye. Netkit: a software component-based approach to programmable networking. *SIGCOMM Comput. Commun. Rev.*, 33(5):55–66, 2003.

[10] Mihai-Lucian Cristea, Willem de Bruijn, and Herbert Bos. Fpl-3: Towards language support for distributed packet processing. In *NETWORKING 2005: 4th International IFIP-TC6 Networking Conference*, 2005.

[11] Robert Ennals, Richard Sharp, and Alan Mycroft. Linear types for packet processing. In *Programming Languages and Systems: 13th European Symposium on Programming*, 2004.

[12] Robert Ennals, Richard Sharp, and Alan Mycroft. Task partitioning for multi-core network processors. In *International Conference on Compiler Construction (CC) 2005*, 2005.

[13] A. E. Eichenberger et al. Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Systems Journal*, 45(1), 2006.

[14] B. Flachs et al. The microarchitecture of the synergistic processor for a cell processor. In *IEEE Journal of Solid-State Circuits (ISSCC)*, 2006.

[15] J. A. Kahle et al. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5), 2005.

[16] J. R. Allen et al. Ibm powernp network processor: Hardware, software, and applications. *IBM Journal of Research and Development: Communication Technologies*, 47(2/3), 2003.

[17] Inc Freescale Semiconductor. C-5 network processor architecture guide. Technical report, 2001.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[19] Ada Gavrilovska. *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processor*. PhD thesis, Georgia Institute of Technology, 2004.

[20] Ada Gavrilovska, Karsten Schwan, Ola Nordstrom, and Hailemelekot Seifu. Network processors as building blocks in overlay networks. In *11th Symposium on High Performance Interconnects*, 2003.

[21] Ada Gavrolovska, Sanjay Kumar, and Karsten Schwan. The excecution of event-action rules on programmable network processors. *1st Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS 2004)*, 2004.

[22] Lal George. Benchmarking μl/μc technology against finely tuned hand coded assembly for the intel ixp 2xxx npu. Technical report, Network Speed Technologies Inc., 2006.

[23] Lal George and Matthias Blume. Taming the ixp network processor. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2003. ACM Press.

[24] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM Press.

[25] ForCES Working Group. Forwarding and control element separation (forces) charter. website, 2005.

[26] Jason Hatashita. An evaluation architecture for a network coprocessor. In *Proceedings of the 2002 IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS)*, Nov 2002.

[27] Erik J. Johnson and Aaron R. Kunze. *IXP2400/2800 Programming: The Complete Microengine Coding Guide*. Intel Press, 2003.

[28] Martin Karsten. SI-WF2Q: WF2Q Approximation with Small Constant Execution Overhead. In *Proceedings of Infocom 2006*, April 2006.

[29] Young-Ho Kim and Jeong-Nyeo Kim. Design of firewall in router using network processor. In *Proceedings of the 7th Internation Conference on Advanced Communication Technology (ICACT2005)*, 2005.

[30] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[31] T. Lewis, H. El-Rewini, J. Chu, P. Fortner, and W. Su. Task grapher: A tool for scheduling parallel program tasks. In *Proceedings of 5th Distributed Memory Computing Conference*, 1990.

[32] Bjrn Liljeqvist and Lars Bengtsson. Grid computing distribution using network processors. In *Proceedings of PDCS'02 (14th Parallel and Distributed Computing Conference 2002) Conference*, 2002.

[33] William R. Mark, R. Steven Glanville, Kurt Akeley, and Mark J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. *ACM Trans. Graph.*, 22(3):896–907, 2003.

[34] Michael McCool, Stefanus Du Toit, Tiberiu Popa, Bryan Chan, and Kevin Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.

[35] Sun Microsystems. Javabeans 1.01 specifications, 1997.

[36] Michael Miller. Internetworking equipment design: Unprecedented look-up rates shakes up packet processing, Nov 2001.

[37] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Trans. Netw.*, 1(3):344–357, 1993.

[38] Devashish Paul. The case for a standalone classification processor, Jan 2002.

[39] Sridhar Rajagopal, Bryan A. Jones, and Joseph R. Cavallaro. Task partitioning wireless base-station receiver algorithms on multiple dsps and fpgas, 2000.

[40] M Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 1999 USENIX LISA Systems Administration Conference*, 1999.

[41] Randi J. Rost. *OpenGL Shading Language (2nd Edition.* Addison-Wesley, 2006.

[42] Niraj Shah, William Plishker, Kaushik Ravindran, and Kurt Keutzer. Np-click: A productive software development approach for network processors. *IEEE Micro*, 24(5):45–54, 2004.

[43] M. Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. In *SIGCOMM '95: Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 231–242, New York, NY, USA, 1995. ACM Press.

[44] Subhash Suri, George Varghese, and Girish Chandranmenon. Leap forward virtual clock: a new fair queuing scheme with guaranteed delays and throughput fairness. In *PODC '97: Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, page 281, New York, NY, USA, 1997. ACM Press.

[45] Agere Systems. The case for a classification language. Technical report, 2003.

[46] Agere Systems. Agere payloadplus network processors, 2005.

[47] Cisco Systems. Cisco 7500 series router datasheet. Technical report, 2002.

[48] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: efficient data structures for implementing a timer facility. *IEEE/ACM Trans. Netw.*, 5(6):824–834, 1997.

[49] Tom Verdickt, Wim Van de Meerssche, and Koert Vlaeminck. Modeling the performance of a nat/firewall network service for the ixp2400. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 137–144, New York, NY, USA, 2005. ACM Press.

[50] Peng Wu, Alexandre E. Eichenberger, and Amy Wang. Efficient simd code generation for runtime alignment and length conversion. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 153–164, Washington, DC, USA, 2005. IEEE Computer Society.

[51] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

[52] T. Yamada, N. Wakamiya, M. Murata, and H. Miyahara. Implementation and evaluation of video-quality adjustment for heterogeneous video multicast. In *Proceedings of the 8th Asia-Pacific Conference on Communications (APCC)*, pages 454–457, 2002.

[53] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. In *SIGCOMM '90: Proceedings of the ACM symposium on Communications architectures & protocols*, pages 19–29, New York, NY, USA, 1990. ACM Press.