Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2002-17

2002-05-03

Design and Performance of Scalable High-Performance Programmable Routers - Doctoral Dissertation, August 2002

Tilman Wolf

The flexibility to adapt to new services and protocols without changes in the underlying hardware is and will increasingly be a key requirement for advanced networks. Introducing a processing component into the data path of routers and implementing packet processing in software provides this ability. In such a programmable router, a powerful processing infrastructure is necessary to achieve to level of performance that is comparable to custom silicon-based routers and to demonstrate the feasibility of this approach. This work aims at the general design of such programmable routers and, specifically, at the design and performance analysis of the processing... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Wolf, Tilman, "Design and Performance of Scalable High-Performance Programmable Routers - Doctoral Dissertation, August 2002" Report Number: WUCSE-2002-17 (2002). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/1135

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/1135

Design and Performance of Scalable High-Performance Programmable Routers - Doctoral Dissertation, August 2002

Tilman Wolf

Complete Abstract:

The flexibility to adapt to new services and protocols without changes in the underlying hardware is and will increasingly be a key requirement for advanced networks. Introducing a processing component into the data path of routers and implementing packet processing in software provides this ability. In such a programmable router, a powerful processing infrastructure is necessary to achieve to level of performance that is comparable to custom silicon-based routers and to demonstrate the feasibility of this approach. This work aims at the general design of such programmable routers and, specifically, at the design and performance analysis of the processing subsystem. The necessity of programmable routers is motivated, and a router design is proposed. Based on the design, a general performance model is developed and quantitatively evaluated using a new network processor benchmark. Operational challenges, like scheduling of packets to processing engines, are addressed, and novel algorithms are presented. The results of this work give qualitative and quantitative insights into this new domain that combines issues from networking, computer architecture, and system design.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2002-17

Design and Performance of Scalable High-Performance Programmable Routers - Doctoral Dissertation, August 2002

Authors: Wolf, Tilman

May 3, 2002

Abstract: The flexibility to adapt to new services and protocols without changes in the underlying hardware is and will increasingly be a key requirement for advanced networks. Introducing a processing component into the data path of routers and implementing packet processing in software provides this ability. In such a programmable router, a powerful processing infrastructure is necessary to achieve a level of performance that is comparable to custom silicon-based routers and to demonstrate the feasibility of this approach. This work aims at the general design of such programmable routers and, specifically, at the design and performance analysis of the processing subsystem. The necessity of programmable routers is motivated, and a router design is proposed. Based on the design, a general performance model is developed and quantitatively evaluated using a new network processor benchmark. Operational challenges, like scheduling of packets to processing engines, are addressed, and novel algorithms

SEVER INSTITUTE OF TECHNOLOGY

DOCTOR OF SCIENCE DEGREE

DISSERTATION ACCEPTANCE

(To be the first page of each copy of the dissertation)

DATE: May 3, 2002

STUDENT'S NAME: Tilman Wolf

This student's dissertation, entitled *Design and Performance of Scalable High-Performance Programmable Routers* has been examined by the undersigned committee of five faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Doctor of Science.

APPROVAL: _____ Chairman

_____

_____

_____

_____

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DESIGN AND PERFORMANCE OF SCALABLE HIGH-PERFORMANCE PROGRAMMABLE ROUTERS

by

Tilman Wolf, M.S.

Prepared under the direction of Professor Jonathan S. Turner

A dissertation presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Doctor of Science

August, 2002

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

DESIGN AND PERFORMANCE OF SCALABLE HIGH-PERFORMANCE PROGRAMMABLE ROUTERS

by Tilman Wolf

ADVISOR: Professor Jonathan S. Turner

August, 2002
Saint Louis, Missouri

The flexibility to adapt to new services and protocols without changes in the underlying hardware is and will increasingly be a key requirement for advanced networks. Introducing a processing component into the data path of routers and implementing packet processing in software provides this ability. In such a programmable router, a powerful processing infrastructure is necessary to achieve a level of performance that is comparable to custom silicon-based routers and to demonstrate the feasibility of this approach. This work aims at the general design of such programmable routers and, specifically, at the design and performance analysis of the processing subsystem. The necessity of programmable routers is motivated, and a router design is proposed. Based on the design, a general performance model is developed and quantitatively evaluated using a new network processor benchmark. Operational challenges, like scheduling of packets to processing engines, are addressed, and novel algorithms are presented. The results of this work give qualitative and quantitative insights into this new domain that combines issues from networking, computer architecture, and system design.

copyright by

Tilman Wolf

2002

to my family

Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my thanks to the numerous people who have helped me in completing this work. First and foremost, I want to thank my advisor Jon Turner for his guidance and the many fruitful discussions. His constant quest for meaningful research and results has taught me invaluable lessons for my career. I also want to thank the members of my thesis committee, Mark Franklin, Jason Fritts, David Richard, and Marcel Waldvogel, who have given me constructive input. In particular, I want to thank Mark Franklin with whom I have collaborated on much of the research related to computer architecture. For support with every-day lab work I want to thank John DeHart, who helped me with many of the measurements. Thanks to Sumi Choi, Dan Decasper, and John DeHart for all the work on the ANN project. For interesting discussions on all kinds of topics, I would like to thank Anshul Kantawala, David Taylor, Samphel Norden, Jai Ramamirtham, Prashant Pappu, and Sherlia Shi. Finally, I want to thank Ed Spitznagel, who was a great officemate for the past four years.

For financial support, I would like to thank DARPA and IBM Research, who paid for my stipend and many conference trips. Especially, I would like to thank Mahmoud Naghshineh from IBM, who sponsored my IBM Research Fellowship.

I thank my fianceé Ana Lucia Caicedo for her love and patience throughout my time as a graduate student. For further non-technical support, I want to thank all the "fools" from the 4^{th} floor in Bryan Hall who were fun to interact with. I thank Dr. Roman for his advice on how to run the Friday Happy Hour and on how to find a faculty position. Thanks to the people from the Evolutionary Biology program at WashU who gave my mind a respite from computer science when needed.

Finally, I thank my parents for having me brought up to appreciate the value of friends, travel, respect, and education.

Tilman Wolf

Washington University in Saint Louis
August 2002

Preface

Hundreds of scientific papers are being published every year proposing changes to existing protocols (e.g., TCP) or introducing new communication mechanisms (e.g., mutlicast, QoS). In practice, however, only very few modifications to the current Internet are deployed. One reason is that most improvements require changes to current Internet routers, which means that expensive equipment that is already in place has to be replaced. For economic reasons, it is infeasible to do this more often than every few years.

To address this problem, it was proposed that packet processing be performed in software on each node in the network. Such a network is called an "active network" or "programmable network." Much funding by DARPA has been put into this research area, including the Active Network Node project at Washington University. One key issue in our project was the limited amount of processing that can be performed at link rates. From this need for more processing power, my work on high-performance programmable routers has been motivated.

The processing engine of a programmable router, the "network processor," is the most performance-critical component. While network processors have recently been developed commercially, not much effort has been put into systematically evaluating the processing requirements and design alternatives of these multiprocessors. It is imperative to obtain a quantitative understanding of system issues in order to develop programmable routers that scale well and can keep up with the ever-growing link speeds and application demands. My work proposes such a programmable router design based on workload measurements and analytic performance modelling. The scope of this work also includes network processor scheduling and system simulation, which brings together issues from networking, computer architecture, and system design.

Chapter 1

Introduction

Industrialized countries have become more dependent on a widely accessible and high-performance networking infrastructure. The Internet is used for personal and business communications as well as online commerce in the form of e-shopping and e-business. It is crucial that networking technology maintains and further improves the performance and the range of functionality of the Internet.

With the wide deployment of optical fiber over the past years, raw bandwidth has become a widely available commodity. A key challenge now is to make use of this bandwidth and extend networks to provide advanced services. The need for new services lies in the fact that the protocols that were originally developed for the Internet are not addressing issues that have recently become critical (e.g., security). It is important that the Internet be able to adapt to support such changes. However, current routers in the network are specialized for fast processing of existing protocols. New protocols can only be supported by replacing existing routers with new routers.

Programmable routers offer a solution to providing this necessary flexibility in the network. By processing packets in software rather than specialized hardware, a programmable router can be re-programmed to support new protocols and services. One key challenge is to design such a programmable router in a way that it can achieve data forwarding performance that is comparable to traditional networks. This dissertation addresses this issue and proposes a scalable, high-performance design for programmable routers.

1.1 Trends in Networking

One reason for the lack of flexibility in today's Internet is that it evolved from a network that was originally designed in the 1970's. The design goals then were to have a simple, packet-switched communication infrastructure, which connects a large number of networks with gateways (or "routers") [Cla88]. The network itself was kept relatively simple and provided basic communication between the end-systems. This led to networking protocols, where most of the complexity is implemented on the end-systems (e.g., retransmission of lost packets, congestion control based on round-trip time measurements).

Over time, several additions have been proposed and implemented in the Internet, because the initial design did not consider them. Since the Internet does not support the dynamic deployment of new protocols, these additions were specifically added to later generations of routers. The following list highlights a few, which are characterized by the need of support by the network (i.e., they cannot be implemented on end-systems only):

- Random Early Detection [FJ93] is a queue management scheme for routers to fairly drop packets from rogue TCP flows. This is can be implemented in a very simple fashion, but it constitutes a type of processing on a router. Almost all current routers implement some form of RED, but only few use it in practice.

- Firewalls [Mog89] are a standard security component of most networks. Packets are filtered depending on rules defined by the network administrator. This enables the blocking of network traffic that could compromise the security of hosts on the network (e.g., port scanning). The firewall rules can be numerous and complex, which requires significant computational power on the firewall to keep up with typical access link speeds.

- Network address translators (NAT [EF94]) are another common component in IP networks. A NAT allows multiple hosts in a stub domain to use a single globally unique IP address. IP packets passing between the stub domain and the Internet are modified by the NAT. This reduces the number of IP addresses used by a stub domain and thereby extends the time before all IP addresses are assigned.

- Web switching [AAP+00] is a method of distributing a web server over several physical machines while presenting a single font-end to the outside. Web

switches parse HTTP requests in packets and determine the appropriate server to which to forward the request. Since the HTTP request is sent only after the TCP connection is established, the web switch also has to splice the TCP connection between client and back-end server.

- IP traceback [SPS+01] allows the network to keep state on the traffic that was forwarded and provides the ability to identify the sources of possibly malicious data flows. For this purpose, routers need to compute a hash from the data packet and store it for a possible later audit.

In practice, these changes either have been slowly added to all routers (as for RED) or they were implemented on servers that are connected to the routers (as for NAT and Web Switching). These specialized solutions are working reasonably well for individual problems, but they are limited in that each extension requires its own solutions. This approach is limited as more and more changes are proposed. A few examples of desirable functions that the network should support are shown in Figure 1.1. Also, as the number of devices connected to the Internet increases with time, the diversity of protocols and the range of services will further increase. A more general approach to providing flexibility to adapt to new protocols and services is by means of programmable routers.

1.2 Programmable Networks

A programmable network consists of programmable routers that have general-purpose processing units in their data path. These processing units can be programmed to perform various protocol operations as well as complex payload processing. Unlike traditional routers, deployment of new protocols can be achieved by reprogramming the system rather than exchanging expensive hardware. This programmability of the data plane extends the traditional store-and-forward paradigm of routers to store-process-and-forward. The processing step is where interesting new services and protocols can be integrated into the network.

The philosophy of opening the network to be programmable raises many administrative issues. In particular, safety and security are points of increasing concern. We assume that programmable networks would only allow system administrators and

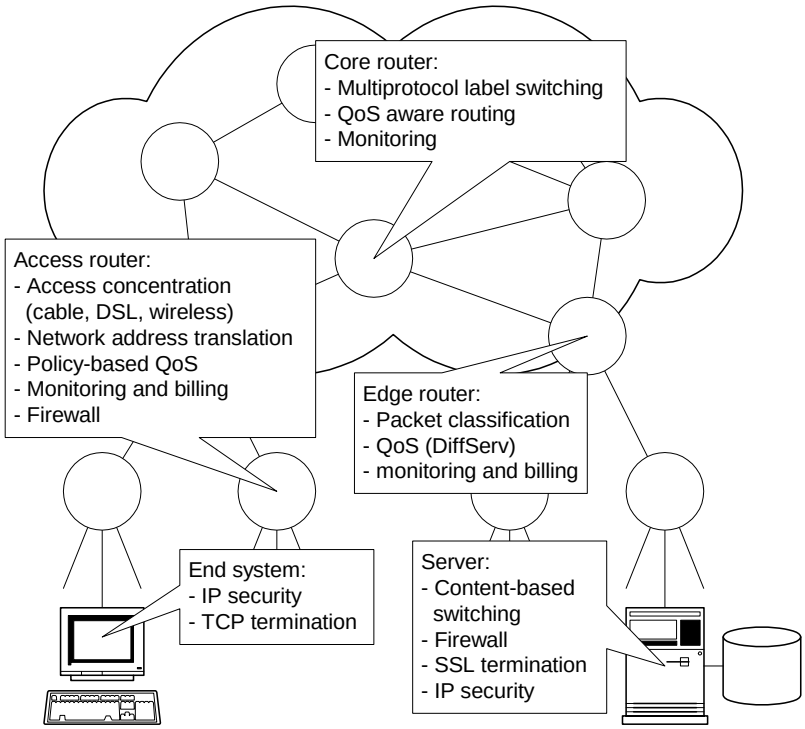


Figure 1.1: Network Services Requiring Programmability.

router vendors to deploy well-tested router software that provides the required service functionality. Such a model might later migrate to a fully open programming platform as proposed by the active networking community.

The flexibility of a programmable router comes of course at a price. Software processing is inherently slower than customized logic that is optimized for protocol processing. However, the increased life-time of a programmable router and the shorter development phase for new protocol support can pay for possibly lower performance. Also, highly parallel processing engines can be developed for this environment to achieve processing rates comparable to traditional routers.

On the commercial side, there has been much development on such network processors. Numerous companies have announced and built such multiprocessor systems-on-a-chip for this environment (e.g., IBM's Power NP, Intel's IXP1200, and Motorola's C-5). However, a general architecture for network processors has not been developed, and a quantitative method for comparing designs is not available. There is much need for a more systematic design approach for these architectures.

The field of programmable routers is still in its early years. It can be expected that there will be much growth in this area. One reason is that the flexibility

provided by programmable routers enables companies to deploy new services faster than in traditional networks. This is particularly important, since the telecommunications market has saturated on raw bandwidth. Services are now what differentiates telecommunications providers, and the ability to quickly react to user demands and competitions will be key to survival in the new telecommunications age.

1.3 Technology Challenges and Opportunities

The need for high performance in the processing engines of programmable routers requires consideration of the newest available integrated circuit technologies. Therefore it is important to observe the technology growth trends in these areas. Of particular interest are:

- Communication Link Speed. Programmable router performance must scale with the growing data link rates.
- Semiconductor Technology. Economic reasons require processing engines to be implemented on single chips. Higher levels of integration allow more processors and memory on such an embedded system.
- Processor Architecture. Advances in processor design give opportunities for higher performing processing engines.
- Application Complexity. With the proliferation of processing engines in networks, application programmers will invent new applications, services, and protocols that will require more and more processing power.

These four areas are relevant to programmable router design as they directly impact the performance requirements (link speeds and application complexity) and the possible solutions (ASIC technology and processor design). In particular, link speed, ASIC technology, and processor performance follow exponential growth characteristics. Moore's "Law" [Moo65] states that the number of transistors on a chip doubles about every 18 months. This exponential growth has impacted other measures in semiconductor technology (i.e., exponential increase of clock speeds). Considering the exponential growth in performance, Figure 1.2 shows the growth rates for the important technologies (each technology is discussed in more detail in Chapter 2, where also the sources for this data are given). It shows that communication

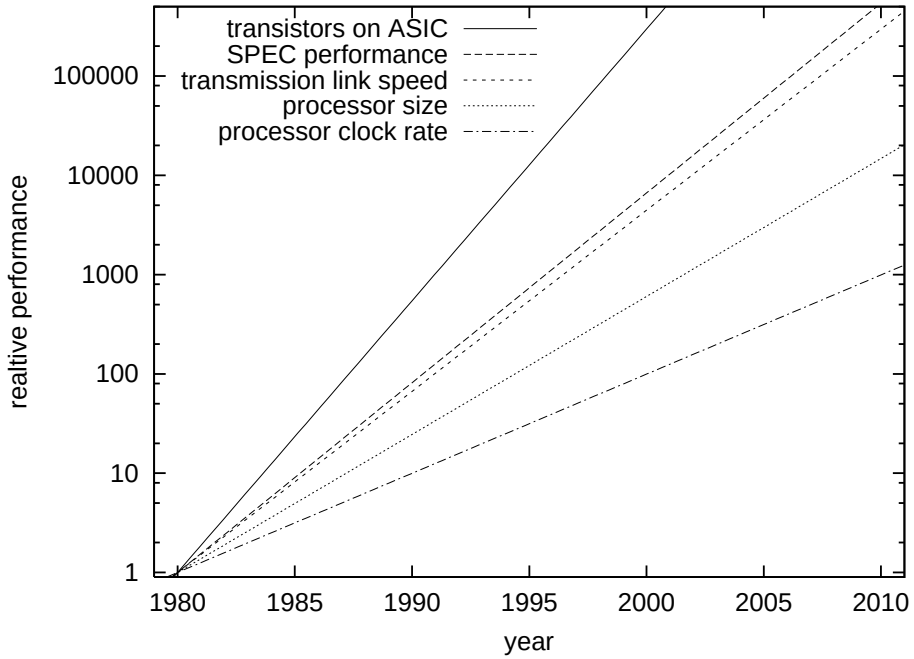


Figure 1.2: Growth of Semiconductor and Communication Technologies.

link speeds grow rapidly, but the density of applications-specific integrated circuits (ASICs) grows even faster. This allows network processor designs with multiple parallel processors to keep up with link speed growth. Since the clock rates and performance of processors themselves grow over time, we can show that processing power of the network processor design proposed in this work even grows faster than link speeds. As a result, more complex network protocols and services can be supported.

1.4 Programmable Router Design

There are two interesting design questions for programmable routers. One is how to provide the functionality of dynamically installing new protocols and services, and the other is how to obtain the necessary performance to compete with traditional routers. The issue of functionality has been addressed by some research in the area. Extending software-based routers to dynamically support new protocols and applications has been a goal of the "active networks" community. An equally important goal for programmable routers is to achieve performance that is comparable to traditional custom-logic routers.

To achieve such performance, two components of a programmable router need to be considered closely. One is the queuing system that moves packets from the

interface to the processors and to the switching fabric is an important component. With the additional processing step, this component becomes more complex than in traditional routers as it has to ensure that the packet is processed by the right instruction code on a processor at the right time. This leads to the question how a processing engine should be integrated in each system and what the data and control path should look like. The other component is the processing engine itself. There are many interesting questions regarding the design and configuration of the network processor. It is necessary to consider realistic workload characteristics in order to obtain results that are meaningful for a particular environment. By using analytic performance models, a broad range of configurations can be considered and general performance trends can be derived.

A related issue is the scheduling of processing on a router port. Since neither traditional processor scheduling schemes nor link bandwidth schedulers apply for this environment, a new scheduling algorithm needs to be developed. The goal of a scheduling algorithm is give performance guarantees for flows and improve the processing throughput of a system by exploiting instruction locality.

In this work, the above issues are discussed and solutions are proposed and evaluated.

1.5 Organization of Dissertation

The dissertation is organized as follows:

Chapter 2 introduces the design that we consider for the programmable router and the network processor chip. Based on the trends in supporting technologies, a scalable design is derived and described in detail.

Chapter 3 discusses the workload characteristics for network processors. Quantitative results are derived from a benchmark that was developed specifically for this purpose. This chapter also discusses how workloads for NPs are different than for workstations and what impact this has on the overall processor design.

The network processor that is described in Chapter 2 has a variety of configuration options. Chapter 4 derives an analytical performance model that can be used to find an optimal configuration and provides a method for exploring the large design space.

The scheduling of the network processor is addressed in Chapter 5. In particular, two scheduling strategies are investigated; one that considers locality in the data

stream to reduce context switching overhead and one that addresses fair sharing of processing resources under the premise of unknown processing times.

Chapter 6 ties the results of Chapters 2-5 together in a system simulation. The analytical results for ASIC optimization and scheduling are simulated and verified. Additional results are derived to evaluate overall system performance and demonstrate its feasibility.

Finally, Chapter 7 summarizes the contributions of this work, addresses future work, and concludes this thesis.

Related work to each topic is discussed individually in each chapter.

Chapter 2

Programmable Router Design

The ability to not only forward packets through a network, but also process them on a router, is the key to implementing new services and protocols without changing the underlying hardware infrastructure. Such processing can range from simple routing and queuing decisions to complex payload modifications. Performing such processing in software rather than custom logic opens the possibility to adapt and deploy new services by simple changes in the software. The crucial challenge in such a system is not only to be able to provide the functionality to dynamically change the forwarding loop for selected data streams. It is equally important that the performance of the system be comparable to custom logic solutions despite the fact that software processing is inherently slower than hardware solutions.

This chapter briefly motivates programmable networks by discussing our implementation of a software-based programmable router. Two applications are introduced to show the spectrum of processing demands necessary. This leads to our programmable router design discussed in Section 2.2. In particular the processing engine is discussed in detail. The scalability of this design is shown under current technology growth trends. Related work at the end of the chapter discusses commercial solutions and contrasts them to the port design proposed here. Parts of this chapter are published in [WT00] and [WT01].

2.1 Software-Based Programmable Routers

General-purpose workstation processors that perform packet routing and forwarding in software were common router configurations in the 1980's. Typical link speeds of a few kilobits per second did not exceed the processing power of such a system. As

performance demands for communication changed in the 1990's towards link speeds of several megabits per second, software-based routers were not able to keep up with this trend. As a result, ASIC-based routers were developed to provide basic protocol processing functionality at high speeds. The majority of current Internet routers are still ASIC-based. Their limitations in supporting new protocols led to efforts re-introduce the flexibility and extensibility of software-based systems.

General-purpose processing as part of the data path and deployment of processing code via the packet itself was initially proposed by Tennenhouse and Wetherall in 1996 [TW96]. From this idea, numerous research projects have spawned to develop infrastructure that can process packets in the data path and dynamically deploy the processing code. The majority of these projects were and are aimed at investigating and implementing software-based "active routers"[1]. Key questions are how to dynamically install protocol processing code in the data path, how to deploy code modules, and how to safely and securely execute arbitrary code on a router. We have developed such a software-based programmable router in the Active Network Node project at Washington University [DPC+99].

2.1.1 Active Network Node

The Active Network Node uses a programmable port processor [DRST01] on the Washington University Gigabit Switch to implement the active processing functionality (see Figure 2.1). The operating system on the port processors is an extension of NetBSD [Net], which allows dynamic installation of code modules (so-called "plugins") into the O/S kernel. IP packets are received by the O/S and processed by the traditional IP processing steps. They are classified, and if active processing is required, they are forwarded to the plugin code modules. There, the special functionality and protocols can be processed. Afterwards, packets are sent through the IP output processing to the switch backplane.

Plugins can be installed dynamically (i.e., when the first packet arrives that requires a new plugin) or statically (i.e., by the administrator for commonly used plugins). The code distribution for this system is done by code servers and clients on

[1]The difference between an "active router" and a programmable router lies in the way packet processing code is developed and deployed. The "active networks" community supports the philosophy of having end-systems (i.e., applications or users) deploy code modules along the path of a data stream. For programmable routers, it is assumed the only well-defined code modules (i.e., new protocol support developed by the router vendor) are installed by administrative entities in the network.

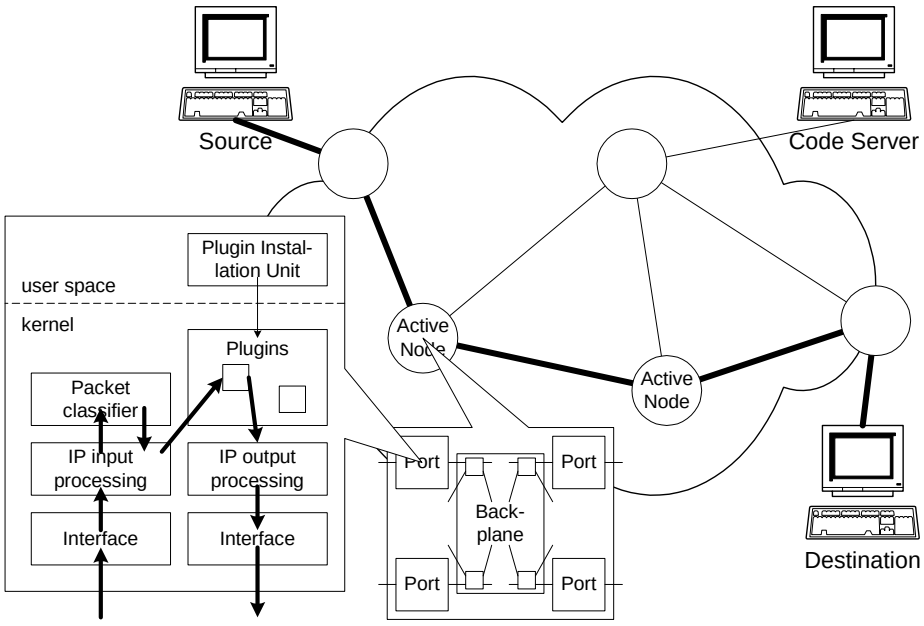


Figure 2.1: Active Network Node. The data path is indicated by arrows.

the respective nodes. If a plugin is requested, the Plugin Installation Unit checks the local plugin repository for the module. If the plugin module cannot be found, it is downloaded from a remote code server.

The major contribution of this project, which makes it different from many other active node implementations, is that the processing of packets occurs entirely in the operating system kernel. This avoids the need to cross the boundary between kernel and user-space, which is associated with overheads from switching between contexts. Therefore much higher packet forwarding rates can be achieved.

2.1.2 Active Applications

The following two examples of applications that use processing on routers illustrate how such a functionality can be used. More applications and their quantitative characteristics are discussed in Chapter 3, where a programmable router benchmark is presented.

Application-Specific Queuing

Once a node is able to consider application-specific requirements of data flows, queuing policies can be adapted to improve the drop policies for flows. In this particular application, which was implemented by Keller *et al.* [KCD+00], a WaveVideo data

flow was used. WaveVideo is a wavelet-based encoding of video data that separates each frame into 33 frequency components [FDW+99]. Low frequency components define the general image in the frame; high frequency components contain the details of the image.

When a WaveVideo transmission encounters congestion in the network, it is desirable that higher-frequency components are dropped first. With traditional queuing, it is not possible for the router to make any distinction among packets. On an active node, a plugin can determine if a packet should be forwarded or dropped. This decision is based on the frequency-level of the current packet and the output queue length. The result of this application-specific queuing is that under congestion the subjective quality of the WaveVideo at the client is significantly higher than under a random dropping policy. Also the quality degrades gracefully under increasing congestion.

While the processing for this kind of application-specific queuing is very simple, it has a considerable effect on the application. It should be noted that this particular method of queuing could not easily be implemented on end-systems in traditional networks without processing capabilities on the router.

Data Aggregation

An example of an active application, which is much more demanding in terms of processing power, is Audio Data Aggregation [WC01]. The basic idea is to use the capability to store and process packets to combine several data flows into a single stream. In the process, data from multiple sources is aggregated, such that the outgoing stream has lower bandwidth that the sum of all incoming streams. This mechanism has also been described as "reverse multicast" [CGM+01].

Practical applications of this aggregation are scenarios, where a large number of sensors transmit periodic status information, for example, a set of temperature sensors in a building. Instead of forwarding all messages from all sensors to one central location, the active nodes in the network can aggregate the information and forward only a "summary." For temperature, a suitable aggregation function is to compute the minimum, maximum, and average temperature of all nodes that are connected to that node. Another example is groups of people, who periodically transmit their geographic position (e.g., soldiers in a battlefield). For this case, the centroid or the bounding box of the group can be used as an aggregation function.

If the networking infrastructure matches the location of the sensors (e.g., all sensors in one room send to one particular active node), then the aggregation steps can be arranged in a hierarchical fashion. By connecting to the output from different levels of the aggregation hierarchy, an observer can navigate the entire sensor space at various levels of detail. Such a hierarchical aggregated multicast is described in more detail in [WC01].

We have implemented this reverse multicast on the Active Network Node for an audio conferencing application. With possibly multiple participants transmitting audio data, the network can bridge the audio information together into a single data stream (with the same bandwidth as an individual stream). There is a twofold benefit in doing that. For one, the overall amount of data that is sent through the network is reduced. Instead of each sender sending to all receivers, the audio is aggregated on the first active node and each receiver receives only one single data stream. The other benefit lies in the reduced processing requirements on the end-system. Since the bridging has already been performed by the network, the audio stream can be played back directly. In a traditional scheme, the end-system would need to bridge all data streams individually. This is particularly useful for "thin clients" (e.g., mobile PDAs), which have limited processing power. Another nice feature of audio aggregation in the network is that the multiparty aspect of the conference becomes completely transparent to the end-system software. In our implementation, we used an IP telephony application, which can only support point-to-point communication. Since the active network performed the bridging function, the aggregated stream was indistinguishable from a point-to-point stream, and thus the application could be used for conferencing without any modifications (except the control plane, which is not considered here).

2.1.3 Performance Issues

The implementations of the above applications have shown that providing the functionality of processing packets in software on a router can be achieved. An interesting issue is the level of performance that can be achieved with such a system. For the application-specific queuing application, very little processing power is necessary, and high throughput can be achieved that is comparable to the throughput of a typical software-based router. However, the processing of the entire payload for the aggregation limits the throughput in the audio application. On the Active Network Node,

which uses a 167MHz Pentium processor, the processing of one 400 byte packet takes in the order of 750μs, which translates to less than 5Mbps throughput. Such performance is definitely several orders of magnitude below the needs in high-bandwidth environments. This low throughput can be attributed to several causes:

- Heavy O/S Overhead. The programmable router uses a full-blown NetBSD operating system, which is not optimized for packet processing.
- Non-Optimized Implementations. The proof-of-concept implementations lack the fine-tuning that can be found in commercial software.
- Non-Optimized Architectures. The processing architecture is not optimized for networking applications, but for traditional workstation-type processing.

These three points clearly limit the performance of a software-based router. However, it cannot be expected that improving on these shortcomings would increase the data throughput by more than an order of magnitude. The real limit of the Active Network Node (and most other active router implementations) lies in the use of a single-processor system. To overcome this limitation, the remainder of this Chapter introduces a programmable router design that emphasizes on a high-performance processing infrastructure.

2.2 High-Performance Programmable Routers

Higher levels of processing power can be achieved by exploiting parallelism in networking workloads. The packet processing can then be performed by highly parallel multiprocessor systems, which are called "network processors." The following discusses the parallelism in networks and shows our router design.

2.2.1 Parallelism in Networking Workloads

In a networking environment, more levels of parallelism are available to the system designer than in traditional workstation multi-processors. This enables the design of processing engines with a large number of independent processors without the need for much communications or synchronization among them. There are three layers at which parallelism can be exploited in a network multiprocessor (see Figure 2.2):

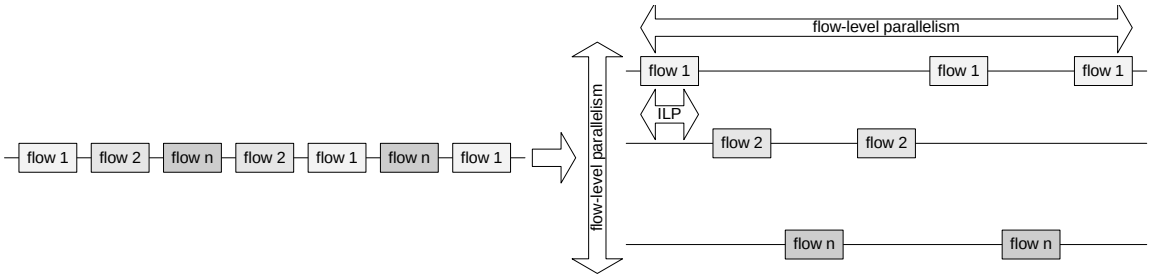


Figure 2.2: Parallelism in Networking Workloads.

- Flow Level. Packets from different flows do typically not interact with each other. Therefore they can be processed completely independently.
- Packet Level. In many protocols, there is no dependency among packets within a flow. For example, simple IP protocol forwarding does not keep or modify state between packets. Therefore packets can be processed in parallel, even if they belong to the same flow. In some cases it is necessary to ensure that the original packet order is restored after processing.
- Instruction Level. When processing a packet, there are several ways that parallelism can be exploited. These approaches are the same as found in traditional processor architecture. In particular, pipelining and instruction-level parallelism can be used.

Parallelism in the processing workload can be directly translated to parallelism in a processing system. For this to work, we need to make a few assumptions. First, it is assumed that the processing requirements for a single flow do not exceed the processing power of a processor. Otherwise processing would have to be split over several processors creating similar synchronization problems as encountered in parallel workstation processors. Second, if there are dependencies (e.g., between packets of a flow), the scheduler assigns packets from the same flow to the same processors. This is discussed in more detail in Chapter 5. These limitations are not expected to pose significant constraints on realistic systems.

2.2.2 Programmable Router Design

The proposed programmable router design extends a traditional router design by adding a network processors on router ports. This differs slightly from some commercial approaches, where the entire router port functionality is implemented on network

processors. In our system, the network processor could be removed and the remaining components would still provide basic router functions.

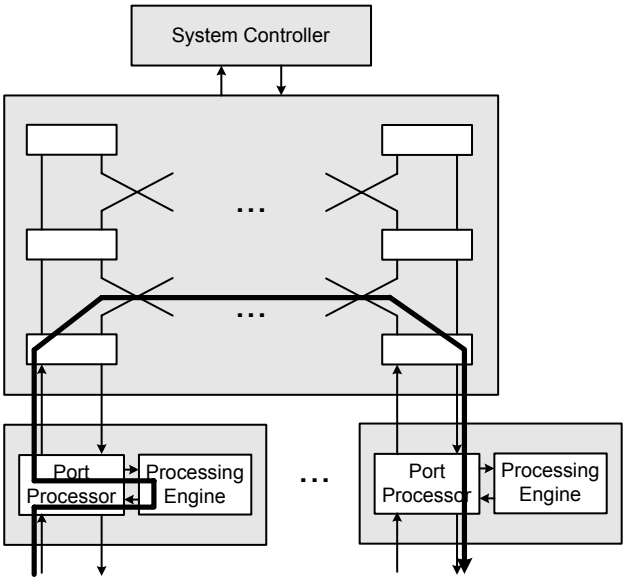
Processing Resources

A traditional router can be augmented to support flexible processing in different ways. One approach is to add a processing engine at each router port. Another is to provide a shared pool of processing engines that can be used to process traffic from any port. These two baseline system designs are shown in Figure 2.3. The routers are based on a scalable cell switching fabric which connects to external links through *Port Processors*. In the first design, all ports are augmented by a *Processing Engine* that can perform active processing. In the second design, a set of router ports is dedicated to active processing. These ports are equipped with processing engines but do not have external interfaces.

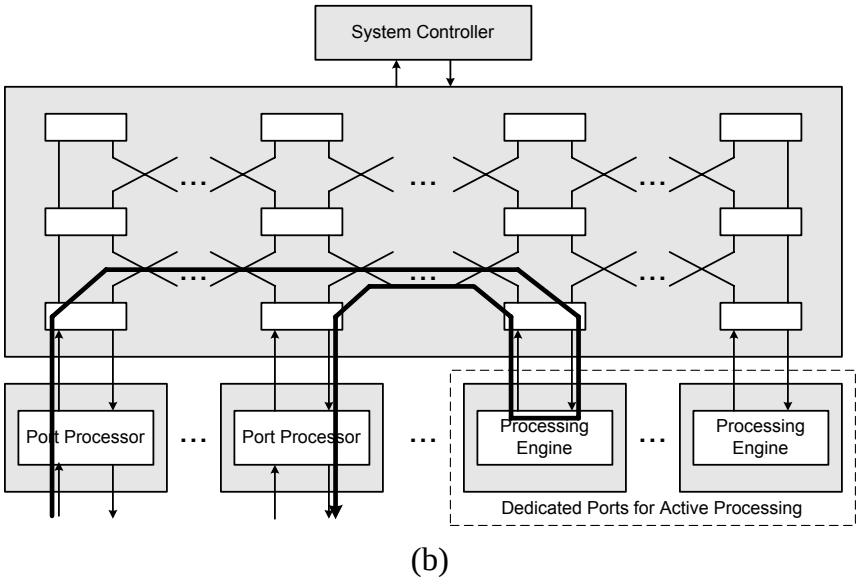
The first approach is most appropriate when all ports have comparable requirements for active processing. The second makes sense when ports have widely varying needs. One can also combine these approaches by having both per-port processing engines and a shared pool to augment the processing power of ports with particularly high processing needs. For the rest of the discussion, the first configuration is assumed.

Packets can be processed on the input port, the output port, or both. We assume processing is performed on the input side. The advantage of this approach is that the processing requirements are limited to the link speed of the connected link. If multicast is used, processing has to be done only once. On the other hand, the drawback is that in case of congestion on the output port, processing might be performed on packets that are later dropped.

Packets belonging to *passive flows* (that is, flows that do not require active processing), are passed directly from the input port at which they first arrive to the output port where they are to be forwarded. Such packets encounter no added overhead or delay, compared to a conventional router. Packets belonging to *active flows* are received by the input port and sent to a processing engine (either on the same port or on a dedicated processing port) where they are enqueued and eventually processed. After processing, the packets are forwarded to the proper output port. If there are processing engines on all ports, processing may also be done at the output port. To provide the maximum flexibility, an input port can distribute packets to various processing engines to achieve system-wide load balancing.



(a) Processing Engines on all Ports



(b) Dedicated Processing Ports

Figure 2.3: System Organization of Programmable Router.

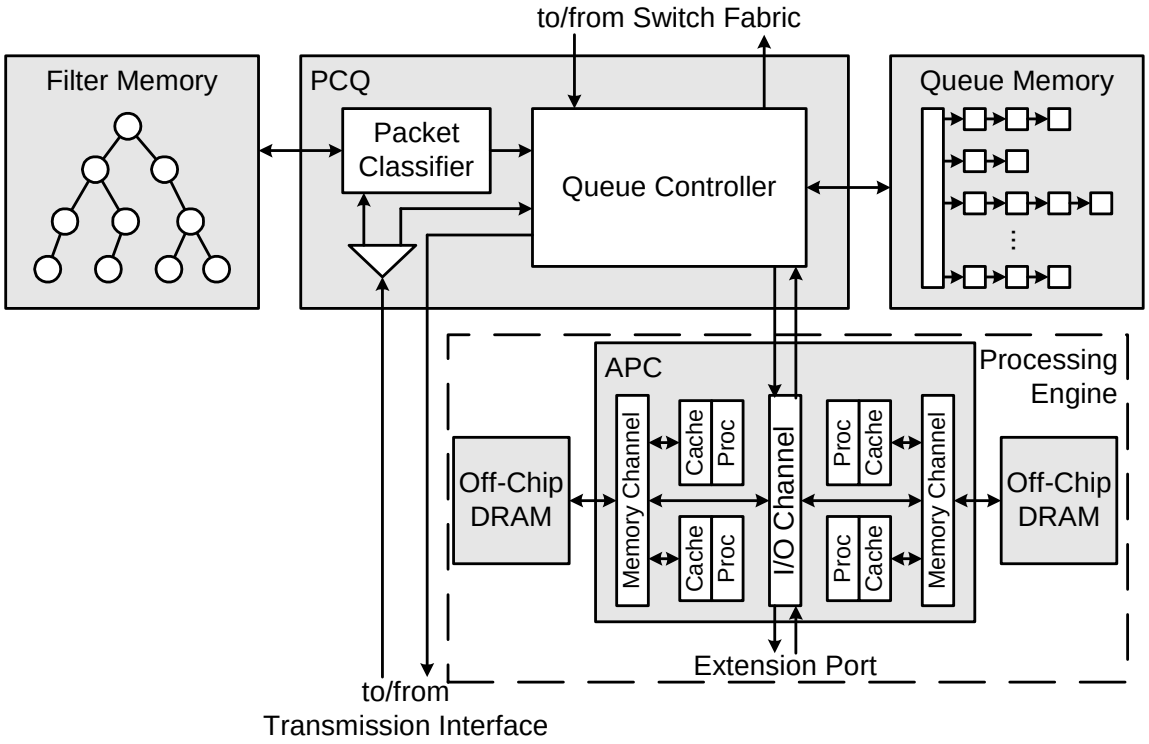


Figure 2.4: Router Port Processor Design. Shaded areas indicate distinct chips.

The switching fabric can be implemented in a variety of ways. For concreteness, we assume a multistage network such as described in [CFFT97]. That system supports external link rates up to 2.4 Gb/s and can be configured to support hundreds or even thousands of such ports. The active router's port processors perform packet classification, active processing and fair queuing. The *System Controller* provides a control and management interface to the outside world and implements routing algorithms and other high level operations.

Router Port

The port design for our programmable router is shown in Figure 2.4. The shaded areas indicate physically distinct logic and memory chips. The center of the port is the *Packet Classification and Queuing chip* (PCQ), which controls the data flow through the port. The processing engine with the *Application Processing Chip* (APC) and off-chip memories comprises the network processor of the system.

Data Path

The PCQ performs classification of packets arriving from the *Transmission Interface*, to determine how they are to be processed and where they are to be sent. It also manages queues on both the input and output sides of the system. The PCQ has two memory interfaces, one to a *Filter Memory* used for packet classification and one to a *Queue Memory* used to store packets awaiting processing or transmission.

As packets are received from the Transmission Interface, the headers are passed to the *Packet Classifier* which performs flow classification and assigns a tag to the packet. At the same time, the entire packet is passed to the *Queue Controller* (QCTL) which segments the packet into cells and adds it to the appropriate queue. This requires a fast general flow classification algorithm, such as the one described in [SSV99].

The processing engine is the key component of the programmable router. In our system, the APC performs active processing for packets that require more processing than just plain forwarding. Note that removing the APC from the system still leaves a router port that is capable of plain forwarding of packets. After processing, packets are queued by the QCTL and then scheduled for forwarding through the switching fabric to the output port. Packets that are received from the switch fabric are also queued, possibly processed, and scheduled for transmission on the outgoing link.

Queuing System

The queuing system of the router stores packets which need to wait for processing or transmission to the switch fabric or the outgoing link. Packets can be assigned to queues in a fully flexible fashion (e.g., per flow or aggregate). The queues can be rate-controlled to provide guaranteed quality of service. The filter database determines whether flows are aggregated or handled separately.

To make the implementation of the queuing system efficient, the queues can be implemented with a combination of two types of memory. Fast (and expensive) SRAM and cheaper (but slower) DRAM. There is also a limited amount of on-chip memory available in the PCQ. The queue data structures can be split onto memories in the following way with respect to the need for fast access versus more storage:

- Per-flow Queue Head and Tail Pointers. These pointers should be stored on the PCQ, since they are accessed frequently. It can be assumed that there is only a limited number of such pointers necessary (particularly since multiple flows can be aggregated into traffic classes).

- Packet Meta Information and DRAM Pointers. This data structure is necessary for each packet and should be stored in SRAM. It contains higher-level information for each packet (i.e., packet size, classification results, output port, etc.) and a pointer to the first memory location in DRAM that contains the actual packet data.

- Actual Packet. The packet is stored in DRAM in a set of "chunks." Each chunk contains a fixed amount of packet data and a pointer to the next data chunk.

Chapter 6 contains more details on the actual implementation of the queuing system on our system simulator.

One key design variable for any router is the amount of memory to provide for queues and how to use that memory to best effect. The usual rule of thumb is that the buffer size should be at least equal to the bandwidth of the link times the expected round trip time for packets going through the network. For 2.4 Gb/s links in wide area networks, this leads to buffer dimensions of roughly 100 MB. Such large buffers are needed in IP networks because of the synchronous oscillations in network traffic produced by TCP flow control and the long time constants associated with these oscillations. In the context of large buffers, per flow queuing and sophisticated queuing algorithms are needed to ensure fairness and/or provide quality of service. Flow control is also needed within a router which has hundreds of high speed ports. Without flow control, output links can experience overloads that are severe enough to cause congestion within the switch fabric, interfering with traffic destined for uncongested outputs. Fortunately, the large buffers required by routers make it possible for cross-switch flow control to be implemented with a relatively coarse time granularity (1-10 ms). Using explicit rate control, output PPs can regulate the rate at which different input PPs send them traffic so as to avoid exceeding the bandwidth of the interface between the switch fabric and the output PP. By adjusting the rates in response to periodic rate adjustment requests from the input PPs, the output PPs can provide fair access to the output links on a system-wide basis or can allocate the bandwidth so as to satisfy quality of service guarantees. Such a rate control mechanism is described in [KDK+02].

2.3 Processing System

The Application Processing Chip provides the general purpose computational resources needed to implement active networking applications. Since these networking tasks are relatively simple and there are many that can be processed in parallel, it is suitable to use very simple processing cores (a more detailed discussion on the choice of processor cores can be found below). Each APC also has several external memory interfaces, providing access to additional memory, which is shared by the processors on the chip. The processors are arranged in clusters, where processors in a cluster share one off-chip memory interface. The APC processors retrieve active packets from queue memory through the *I/O Channel*, process them, and write them back out to the proper outgoing queue. Processing instructions and flow state information are stored in the on-chip and off-chip memory. The scheduling of these processing engines and issues related to maintaining consistency in flow state is discussed in Chapter 5.

The APC design shown in Figure 2.4 contains four *Application Processing Units*. Each application processing unit consists of a processor, an SRAM cache memory for instructions and data, and a memory controller for off-chip DRAM access and communication with the I/O channel (not shown). The processing units are linked to the PCQ through an I/O channel, which also provides the interface to the extension port. The *Memory Channel* provides access to the external DRAM.

2.3.1 Processor Architectures

There is a wide range of processor architectures that can be considered for processing on a programmable router. The key criteria that have to be considered for selecting a suitable processor are:

- General-Purpose Processing Capability. The key to flexibility to support new protocols and services lies in having general-purpose processing engines. This does not exclude a design that has a few special hardware accelerators for speeding up common tasks (e.g., checksum computation, table lookups).

- Small Physical Size. One major constraint for processing engines is that they need to fit onto a single chip. Due to the high level of parallelism that can be exploited, multiple small processors can provide more performance than a single, more sophisticated processor.

- Performance on Networking Tasks. The characteristics of tasks that process packets on routers is significantly different from traditional workstation tasks (a quantitative comparison of our network processor benchmark with a workstation benchmark shows this in Chapter 3).

For the processing engine of our router, we consider RISC cores, VLIW processors, DSPs, and specialized co-processors.

Reduced Instruction Set Computers

Reduced Instruction Set Computers (RISC) [Pat85] were proposed in the 1980's as an alternative to increasingly complex "Complex Instruction Set Computers" (CISC). RISC architectures are typically relatively simple (several dozen to few hundred instruction, orthogonal addressing modes, etc.), pipelined, and supported by a sophisticated compiler. Since some of the complexity of execution is off-loaded to the compiler, the RISC processor itself is simpler than a CISC processor and thus can be implemented in a smaller area. RISC architectures are particularly suitable for embedded systems, where area constraints are much more pressing than in workstation environments.

It is important to consider the processing performance that can be obtained from a processor architecture versus the size that it occupies on the chip. Figure 2.5 shows this tradeoff for several generations of the most common processor architectures. The performance metric (y-axis) is the processor's performance on the SPEC benchmark [Sta95][2] normalized by the clock rate of the processor. This allows a comparison of various processor generations without considering improvements in clock rates due to smaller feature sizes (as done in [AHKB00]). The performance and size values are obtained from [CPU].

As can be seen, the performance of a processor does not increase linearly with the area it occupies. Instead, larger processors of newer generations show proportionally lower performance than smaller, simpler processors. This is partly due to fact that these architectures are optimized for processing of single tasks. Traditional workstation workloads cannot easily be parallelized as is possible in networking environments. High levels of parallelism can be exploited much better by multiple simpler processors, which results in linear performance improvements when replicating processors. Another reason for the observed trend is that modern processors are

[2]The SPEC performance values were adjusted to SPEC CPU92, because certain architectures predate the release of the SPEC CPU95 benchmark.

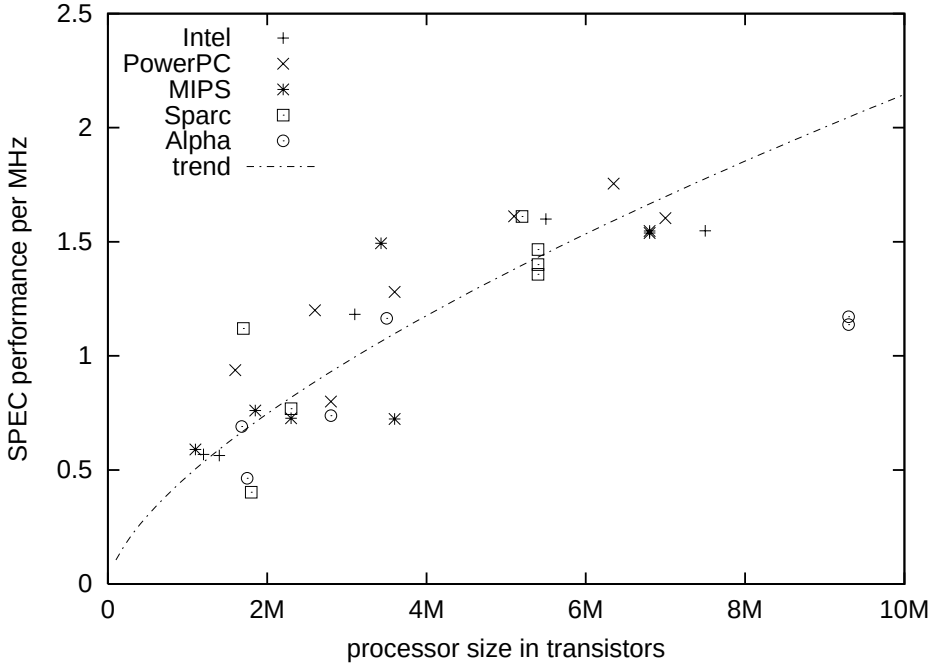


Figure 2.5: Processor Performance in Relation to Complexity. The performance metric is SPEC rating per MHz and the complexity is the processor size in transistors.

optimized for speed at the cost of chip area. It is not necessarily clear that older, simpler architectures can be implemented on modern ASICs and clocked at gigahertz rates. However, the clock rates considered in network processor are not at the cutting edge of technology as they are for workstation processors. Therefore, it can be assumed that most processor implementations can be adapted to these increasing clock rates.

This leads to the conclusion that simple RISC cores, like MIPS [MIP98], ARM [ARM99], and PowerPC [IBM98], are suitable processor architectures for network processors.

Superscalar and Multithreaded Processors

Numerous improvements to the basic RISC architecture have been proposed. In particular, several methods for exploiting parallelism on the instruction level and thread level have been proposed:

- Superscalar and Very-Long Instruction Word Processors. These processors can issue and execute multiple instructions per cycle exploiting instruction-level parallelism (ILP). Superscalar processors can be scheduled statically by the

compiler or dynamically using scoreboarding techniques. VLIW can only be scheduled statically by the compiler.

- Fine-Grained Multithreaded Processors. Such multithreaded processors maintain hardware contexts for several threads. In case of a stall of a thread, the processor can continue processing another thread [ALKK90]. Most recent multithreaded architecture support zero-overhead context switching, which means that no cycles get lost during a context switch [MMC00].

- Simultaneous Multithreaded Processors. SMT processors also maintain multiple thread contexts. The processor can dynamically schedule multiple instructions from all available threads [HKN+92], [TEL95]. This approach is a combination of superscalar and multithreaded processors.

The main goal of all these architectures is to improve the execution time of a single thread and increase the overall processor utilization. In the context of network processors, the individual thread execution time is less significant than the overall processing throughput of the system. Thus, all the architectures need to be examined in the light of usage in a network processor. Such a study has been done by Crowley *et al.* [CFBB00]. The results have shown that simple RISC multiprocessors perform best under networking workloads. When considering a heavier O/S overhead, the SMT architectures perform best.

For our router design, we chose to use simple RISC processors instead of considering more complex architectures. One key reason is the higher space requirements and smaller overall performance gain as shown in Figure 2.5. Another reason is that with the trends in integrated circuit technology, processor architecture, and link rates, the performance of a RISC multiprocessor outperforms a multiprocessor that uses increasingly complex processor architectures (which will be discussed in more detail in Section 2.4).

The one improvement over RISC that will be considered in this work is the idea of multithreading. Since the shared off-chip memory can create long processor stalls under high loads, it is desirable to keep the processors utilized by adding a few thread contexts. Such an approach is discussed in more detail in Chapter 4, where the performance model of our system is introduced.

Digital Signal Processors

Digital Signal Processors (DSP) are processors that are specialized for performing highly regular and real-time critical processing. A typical DSP consists of several arithmetic units and multipliers. In each cycle, an instruction can be executed on each unit and data can be moved between them. The operation of a DSP is usually highly pipelined, which allows the DSP to operate at high clock frequencies. Multiple memories or peripherals provide the data and instructions for the DSP. With clock rates as high as one GHz, DSPs can provide significantly more performance than general-purpose processors if the applications match well the regular operating patterns of the DSP architecture.

Traditionally, DSPs have been used in applications involving digital control, audio, telephony, imaging, and video. Recently, DSPs have also been developed for use in networking. For example, the C6000 family from Texas Instruments implements a complete TCP/IP stack on the DSP. This allows certain networking equipment to operate without a host processor, which was traditionally required for running the network protocol processing.

For the programmable router design in this work, we do not consider DSPs, because the arbitrary processing requirements of a programmable router cannot always be mapped efficiently to the regular, pipelined operation modes of DSPs. It is conceivable, though, that the APC can be implemented as a hybrid system that is equipped with a set of DSPs for the few applications which execute more efficiently on such processors.

Specialized Coprocessors

There is a set of common operations in the networking domain that are used in a wide range of protocols. For some such operations, specialized coprocessors can perform significantly better than general-purpose RISC processors. Such functions are:

- Checksum and CRC Computations. Several protocols in a typical protocol stack require a checksum or CRC computation across parts of or the entire packet.
- Table Lookup and Packet Classification. Routing and QoS algorithms require such functionality to determine the flow to which a packet belongs.

- Encryption Processing. This is only one example for a processing step that requires intense processing and that can be efficiently implemented in specialized hardware.

As with DSPs, coprocessors can be used in addition to the general-purpose RISC processors. In our system, such coprocessors could be co-located with each RISC processor if they are relatively small (e.g., checksum coprocessors) or be shared among a set of processors (e.g., table lookup unit).

2.3.2 Memory System

For efficient processing of flows, the processors should have enough memory to store both a small operating system kernel and the code for the applications being used. In addition, they need to be able to store the current packet that is being processed as well as per flow state information for the current flow. Since the packets can be brought in from the queue memory as needed, then promptly written back out, not too much on-chip memory is needed for the packets themselves, but the program code and per-flow state could easily consume hundreds of kilobytes of memory. This suggests that the bulk of data should be stored in the off-chip DRAM. To allow the processors to operate at peak efficiency, on-chip SRAM can be used to cache data and instruction code.

Integrated circuits with embedded DRAM have recently been developed. Since the CMOS fabrication process for DRAM optimizes for density rather than speed as in processing logic, it is challenging to combine both on a single chip. We are not considering a second level cache in DRAM at this point, but it might be possible in the future to use DRAM technology and increase the available on-chip memory.

The bandwidth required between an APC and its external memory is determined by the number of APUs on the chip, the instruction-processing rate of those APUs and the fraction of instructions that generate requests to the external memory. Chapter 3 and 4 discuss in detail the application requirements and the performance tradeoffs for different memory interface configurations.

2.3.3 I/O System

The I/O channel on the APC transports data packets from the queue memory to the processors. The required I/O bandwidth is a key consideration. In this design,

the bandwidth required for the interface to/from the PCQ can be bounded by the link bandwidth. For 2.4 Gb/s links, this implies a bandwidth of 300 MB/s in each direction. To allow for loss of efficiency due to packet fragmentation effects (caused by packets being divided into cells) and to reduce contention at this interface, it is advisable to increase the bandwidth at this interface to 1 GB/s. This can be achieved with a 32 bit interface in each direction, operating at a clock rate of 250 MHz, which is feasible in .25 μm technology. It is also possible to implement the I/O Channel as a ring. This can give a simpler implementation but may yield larger delays.

2.3.4 Configurations

The actual configuration of an APC depends on a variety of factors. The issues that influence the performance of a configuration are:

- Workload. Computationally intense workloads require more processing and on-chip memory. Simpler processing requires more I/O bandwidth, because packets are moved more quickly between the queue controller and processors.
- Technology. Advances in technology allow higher levels of integration, which yields more processors and more memory on a chip.
- Power Consumption. With more components on a chip and higher clock rate, the power consumption of the APC becomes a critical issue.
- Cost. The overall size of the APC affects the cost of the router port and the overall router system.

Since the overall size of an APC is limited, there is a tradeoff between placing more processors or more memory onto the APC. More processors mean smaller on-chip caches, which leads to inefficient execution and more traffic on the memory channel. Fewer processors mean larger on-chip caches and more efficient execution, but also limited overall processing power. It is a challenging problem to find an optimal configuration for the APC. Chapter 4 discusses an analytic solution for this optimization problem and shows results for our system.

2.4 Scalability

To illustrate the long-term usefulness of the proposed design, we describe its scalability under current technology trends. The design can be scaled in three dimensions:

- Increase in Number of Router Ports. The number of ports can be increased by configuring the multistage interconnection network to have a larger number of stages. For the design in [CFFT97], a three stage network can support up to 64 ports and has an aggregate capacity of 154 Gb/s, while a five stage network can support up to 512 ports and has an aggregate capacity of 1.2 Tb/s.

- Increase in Processing Capacity Per Port. One can increase (or decrease) the active processing capacity by incorporating more or fewer APC chips at each port. For systems with only a small amount of active processing, APCs can be omitted from most ports, and packets requiring active processing can be forwarded from the ports at which they arrive to one of the ports containing an APC.

- Increase in Processing Capacity per APC. Developments in technology allow more processors, memory, and I/O components to be implemented on a system-on-a-chip. This increases the overall processing power of the APC.

The most significant potential for more processing power lies in the higher levels of integration that can be achieved with newer generations of CMOS technology.

2.4.1 APC Design Scalability

To increase the per-port processing capacity of the router, multiple APCs can be arranged in a daisy-chain configuration that connects the APCs via the extension port on the I/O channel. Each interface that connects to another processing chip acts as a gateway and routes data to other APCs further down in the chain. This design requires that the I/O channel be able to handle the total bandwidth between the queue controller and the APC. It can be assumed that this requirement can be met even for faster link rates. Note, that the total amount of bandwidth between the queue controller and the application processing chips is at most twice the external link bandwidth, since each packet is sent at most once to the processing chips and sent at most once back to the queue controller. Since each processing chip has its own off-chip memory interfaces, the traffic from off-chip memory accesses is restricted to the

individual APC and does not aggregate over multiple chips. Finally, only data traffic that requires processing needs to be sent to an APC. While we expect processing to be an important element of future routers, we expect most packets to be forwarded without processing for the foreseeable future.

2.4.2 Technology Scaling

The trends in technology that are relevant to our design are increases in link speed, processor performance, and the transistor density and clock rates of Application-Specific Integrated Circuits (ASICs). The basis for our analysis is Moore's Law, which states that the number of components on an integrated circuit doubles roughly every eighteen months [Moo65]. This was first predicted in 1965 and has found to be accurate ever since [GGPY89]. It is expected that these trends will continue at least for another decade [Sem01].

While Moore's Law only addresses the number of components on a chip, it has been observed that most technology trends in the semiconductor industry follow similar exponential growth. Therefore we approximate growth trends for technologies relative to the performance in year t_0 by:

$$\text{performance in year } x = a_{t_0} \cdot e^{b \cdot (x - t_0)}. \tag{2.1}$$

The parameters a and b are characteristic for the particular technologies. For the plain interpretation of Moore's law, a is 50 components per chip in 1965 and b is 0.41, which corresponds to doubling every year and a half. Quantifying these parameters for the technologies relevant to the proposed router design, we can obtain a quantitative understanding of its scalability.

Communication Links

The growth of communication link speed is an important factor in router design since it determines how much data has to be processed. Figure 2.6 shows the growth in speed for communication links since 1975 [Rob00] [Chr99]. It is important to note that until the middle '90s, optical link speeds grew at about the same rate as electronic communication links. However, with the advent of Dense Wavelength Division Multiplexing (DWDM), the growth of bandwidth on a single fiber has started growing much faster [Chr99] (not shown in figure). This trend is expected to continue until a

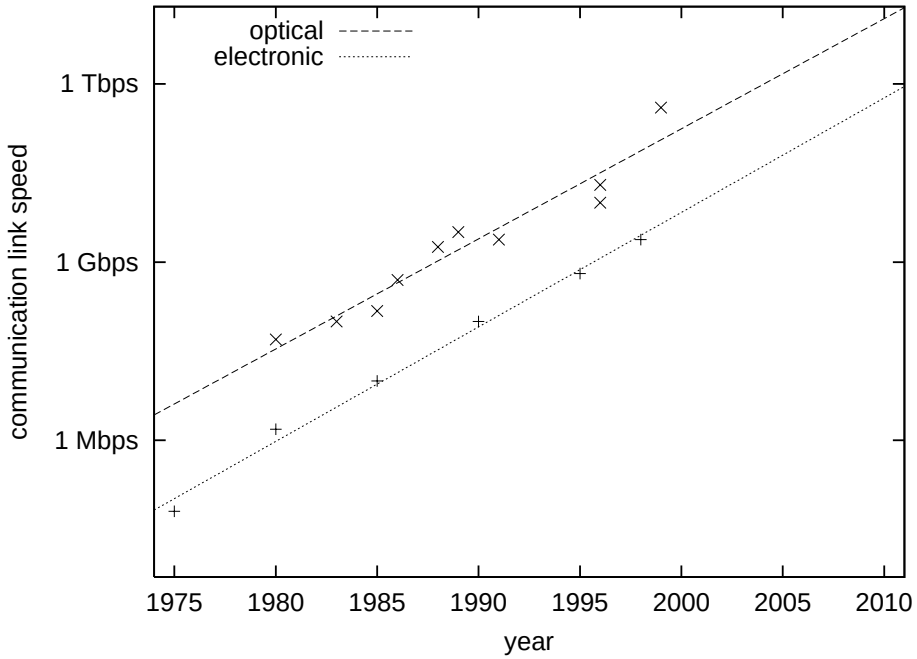


Figure 2.6: Growth of Link Speed for Electronic and Optical Communication Links.

the limit in the communication spectrum of optical fiber is reached. However, each WDM channel must be processed electronically for the foreseeable future regardless of the overall WDM channel count. Therefore, we only consider the growth of a single channel. From the figure, we can extract a value of $b \approx 0.43$.

Processors

Advances in processor design have continuously increased the processing performance of CPUs. Figure 2.7 shows that growth in terms of performance on the SPEC INT92 benchmark. The performance values for the processors were obtained from [CPU]. There are two parts to the improvement in performance. For one, the clock rate at which processors operate increases as shown in Figure 2.8. The other is the improvements in the processor architecture, increase in on-chip cache sizes, etc., which increase the efficiency of processors. This affects the processor size as shown in Figure 2.9. Both contribute to an improvement in SPEC performance. The growth parameters are shown in Table 2.1.

These trends indicate that newer generations provide more processing performance at the cost of larger and more complex circuitry (as discussed above and

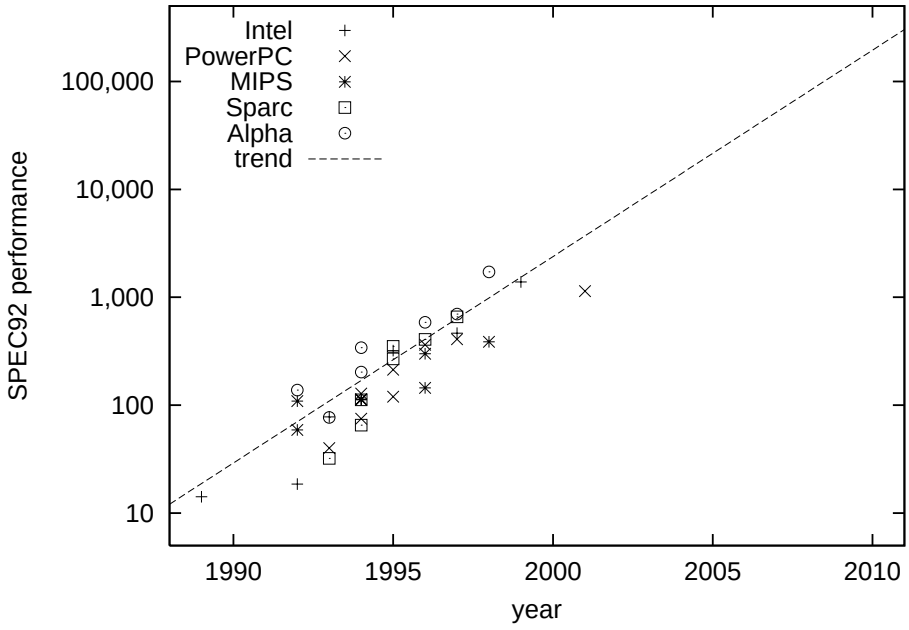


Figure 2.7: Growth of Processor Performance. The results are normalized to the metric used in the SPEC 92 benchmark.

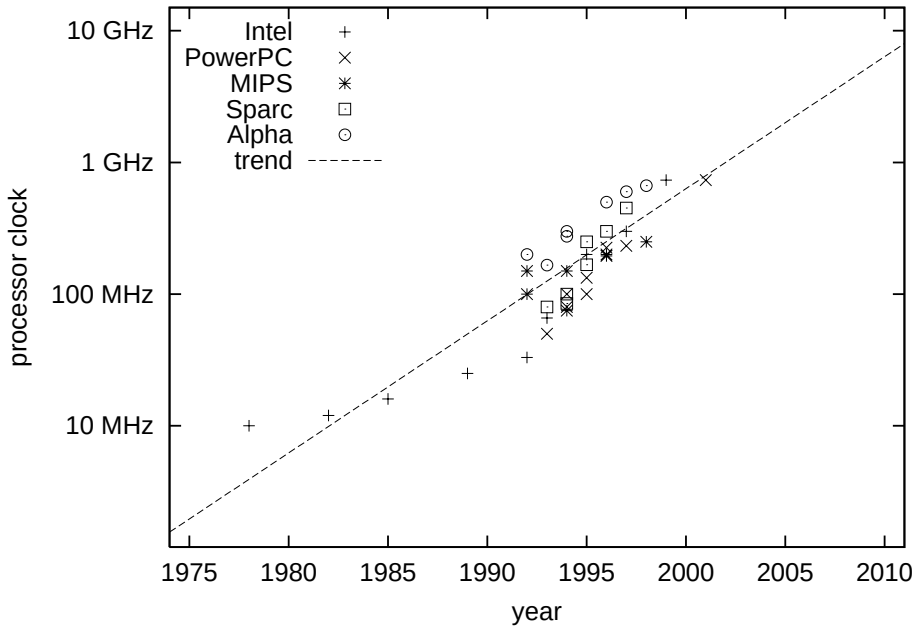


Figure 2.8: Growth of Processor Clock Rate.

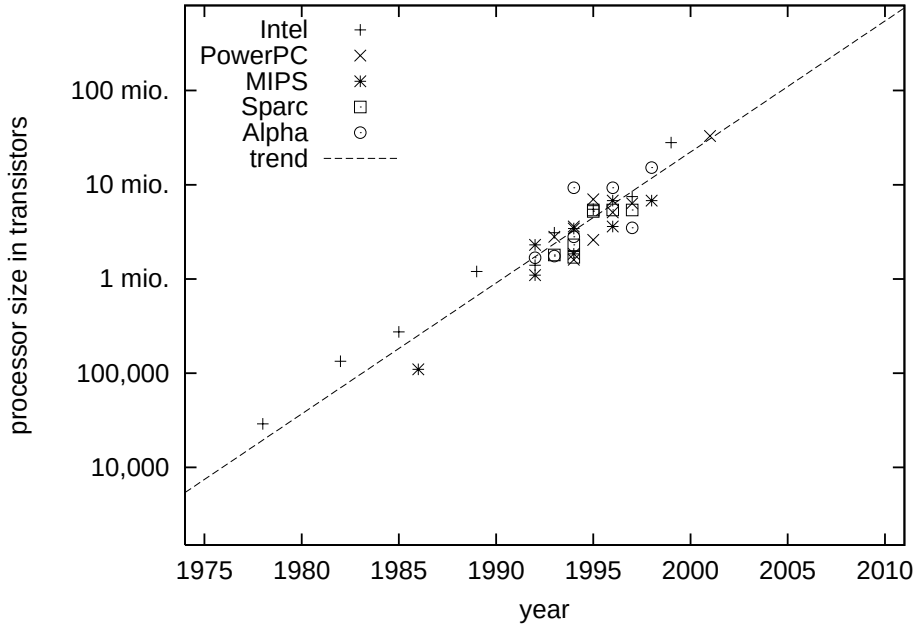


Figure 2.9: Growth of Processor Size. The size is expressed in transistors.

illustrated in Figure 2.5). Below we will see that simple RISC cores are not only currently a better choice for network processors, but the performance gap over complex processors will grow with time (see Figure 2.11.

Application-Specific Integrated Circuits

For economic reasons it is desirable to implement the APC on a single ASIC. Thus, the size of an ASIC poses a limit on how many processors or memories can be combined in an APC. Figure 2.10 shows the number of logic gates that are available on ASICs. One logic gate translates roughly into four transistors (for comparison with Figure 2.9). The values shown correspond to the IBM 5S, 5X, SA-12, SA-12E, SA-27, SA-27E, and Cu-11 families, the LSI 5V, G10-p, G11-p, and G12-p families, and the NEC CB-C7, CB-C8, CB-C9, CB-C10, CB-C11 families.

With quickly decreasing feature sizes and slightly increasing chip sizes, technology provides a lot of potential for powerful multiprocessors. Note that the growth parameters with $b = 0.63$ are slightly more optimistic than Moore's Law predicts.

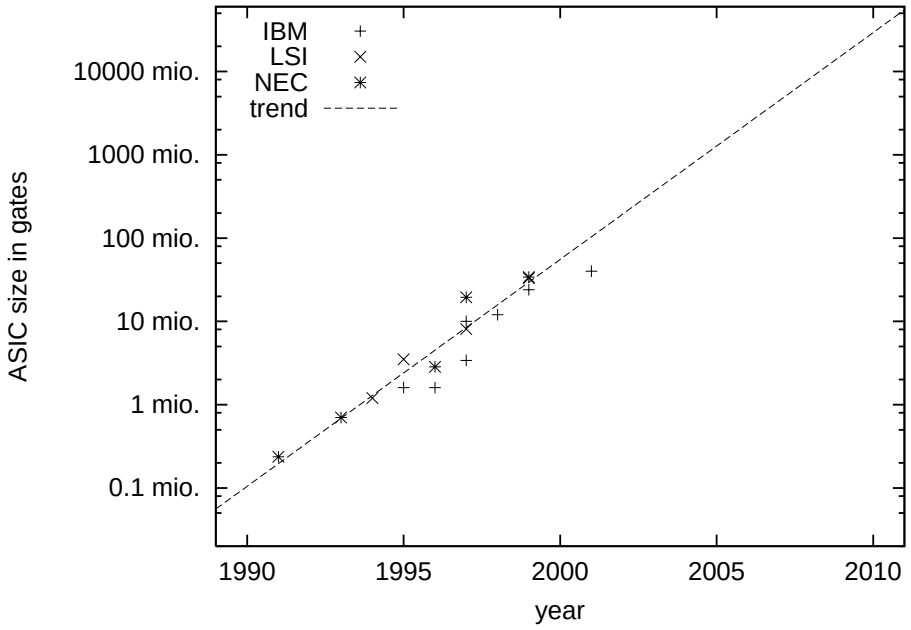


Figure 2.10: Growth of the Number of Available Gates on an ASIC.

Table 2.1: Growth Parameters of Key Technologies. The values for a are normalized to the year 2000. Sizes are given in million transistors (Mtx) and million gates (Mgates) (1 gate $\approx$ 4 transistors).

Technology		a	b	Time to double
Communication	electronic links	6.8 Gbps	0.44	18 months
	optical links	175 Gbps	0.42	19 months
Processor	SPEC performance	2400	0.44	18 months
	clock	630 MHz	0.23	36 months
	size	22 Mtx	0.32	26 months
ASIC	size	55 Mgates	0.63	8 months

Impact on APC Design

The parameters for technology growth can be used to give an estimate on the performance growth of the APC. Assuming that we can continue to arbitrarily parallelize the workload on the APC, we can compare three design approaches:

- Single Processor Design. This corresponds to software-based programmable router implementations.
- Complex Multiprocessor Design. Using the improvements in processor architecture, more complex and more powerful processors can be used to process packets.
- Simple Multiprocessor Design. Following our APC design, the ASIC can be filled with simple RISC cores rather than more complex processors.

Figure 2.11 shows the performance of these three approaches. Performance is expressed as processing power per byte of link data, where the processing power is the SPEC INT92 metric per MHz. The trends are based on results from Table 2.1. It can be seen that the single processor system can just barely keep up with the increasing link rates. Both multiprocessor systems show an increase in processing power over time. The simple multiprocessor outperforms the complex multiprocessor by about one order of magnitude by the year 2010. This shows that the design of the APC is the most promising to provide increasing processing power to support more complex applications.

APC Scaling

To give a concrete example of the scalability of the APC design, Table 2.2 shows possible configurations for the APC for the next decade. The scaling follows roughly the technology trends discussed above. The ASIC scaling is considered by decreasing the feature size by a factor of two every three years. Also, the total APC area increases over time. Since the application complexity can be expected to grow with time, increasing processor caches sizes are expected. Assuming that the number of threads are sufficient to let the processors operate at a high utilization, the processing power can be estimated by multiplying the processor clock rate by the number of processors (in giga-instructions per second (GIPS)). While the performance depends on particular configurations and workload parameters, it provides a rough estimation.

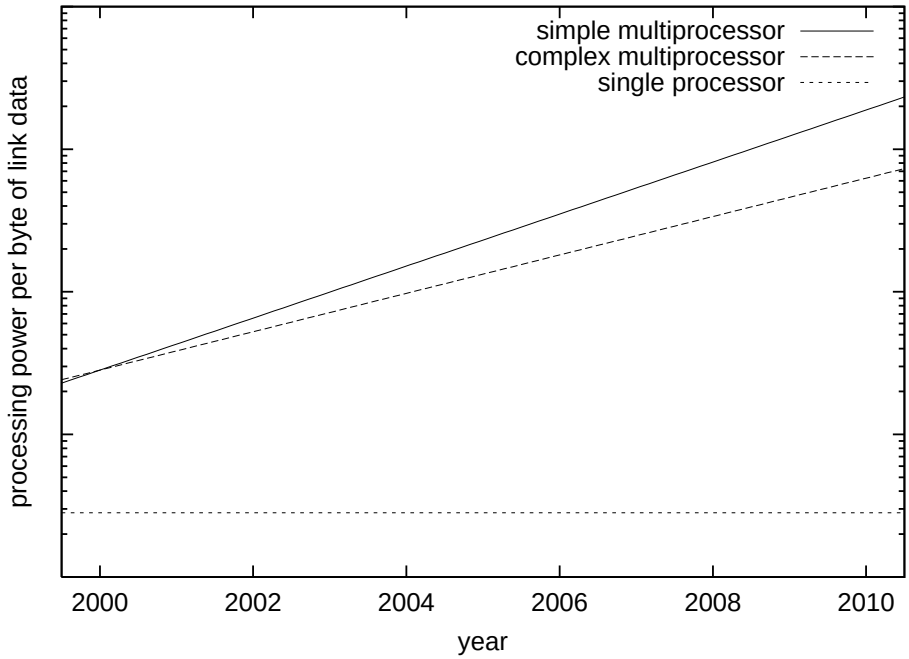


Figure 2.11: Processing Power per Byte of Link Data.

Considering the increasing link speeds, the APC is still able to provide increasing processing performance per byte of link rate. A more detailed performance model is discussed in Chapter 4.

2.5 Related Work

There has been much work on developing software architectures as well as commercial efforts to develop network processors.

2.5.1 Programmable Routers

Software environments for packet processing on programmable routers have received much attention in recent years and many such systems have been developed:

- Tennenhouse *et al.* at MIT proposed the capsule mechanism, where packets carry code fragments, which are interpreted on the nodes in the network [TW96]. The ANTS toolkit is an architecture for dynamic deployment of processing code. Instead of each packet carrying code, only references to code modules are transported. Along the path of the flow, code is installed on demand as it is needed [WGT98].

Table 2.2: Scalability of APC with Improvements in Technology.

Year	1999		2002		2005		2008
Feature size (μm)	0.25	0.18	0.12	0.09	0.06	0.045	0.03
Number of processors	6	12	16	24	30	48	64
Number of threads per proc.	1	1	2	2	2	4	4
I-cache size (kB)	16	16	32	32	64	64	96
D-cache size (kB)	16	16	32	32	64	64	96
Number of memory interfaces	1	2	2	3	3	3	4
Memory interface width (bit)	32	32	32	48	48	64	64
Processor area (mm^2)	1.4	1.0	0.67	0.50	0.33	0.25	0.17
SRAM area per MB (mm^2)	140	100	67	50	33	25	17
Total APC area (mm^2)	87	107	120	139	172	204	248
Processor clock frequency (MHz)	400	550	830	1,100	1,700	2,200	3,300
Processing power (GIPS)	2.4	6.6	13	26	51	106	211
Link rates (Gbps)	1	1.9	3.6	6.9	13	25	47
Instructions per byte of link rate	19	28	29	31	31	34	36

- To improve the performance of ANTS, the Joust project at University of Arizona implemented the Java virtual machine on the Scout operating system [HBB+99]. This implementation is about two to three times faster than off-the-shelf implementations of the Java VM.

- The Smart Packets project at BBN is another capsule approach that aims at network management functions [SJS+99]. Using a compact programming language, small pieces of active code are sent with packets and executed on nodes for diagnostic and management functions.

- The Switchware project at University of Pennsylvania addresses the issues of safe execution of active code by using a very restrictive language (PLAN [HKM+98]) to program capsules [AAH+98]. In addition, more complex functionality is implemented in "switchlets," which can be called by active packets. The switchlets are also programmed in a language that allows formal verification for safe execution.

- To make systems interoperable, a common architecture for active router operating systems has been proposed. This Node Operating System (NodeOS [Pet01]) defines common abstractions for communication and processing resources. On top of the NodeOS, several Execution Environments (EEs) can operate in parallel. Different active functionalities are implemented within the EEs. One instance of a NodeOS-compliant architecture is the CANES and Bowman project

at University of Kentucky and Georgia Institute of Technology [MBC+99], [MBZC00]. CANES is an implementation of the NodeOS and Bowman is an Execution Environment.

Other issues are how to signal control information and reserve resources in the network [CCF+01], and how to safely and securely executed code [MLP+01]. There is a number of other projects, which are discussed in several survey papers on active networks [TSS+97], [CDMK+99], [Pso99].

2.5.2 Network Processors

The promise of flexibility and lower development times have raised considerable interest in network processors within the commercial communications sector. Unlike active networks, today's commercial network processors are mainly optimized for packet header processing of established protocols. Nevertheless, these network processors are general-purpose processing engines that can also be used for more complex applications. The following list of products gives a brief overview of available and publicly announced systems and some of their basic characteristics:

- IBM PowerNP [IBM00]: 8 processing units with 2 processors each, one PowerPC control processor, 133 MHz clock rate, 1.6 GB/s DRAM bandwidth, 8 Gb/s line speed, 2 threads per processor.
- Intel IPX1200 [Int00]: 6 processing engines, one StrongARM control processor, 200 MHz clock rate, 0.8 GB/s DRAM bandwidth, 2.6 Gb/s line speed, 4 threads per processor.
- Lexra NetVortex [Lex00]: 16 processing units, 450 MHz clock rate, 4 threads per processor.
- Lucent Fast Pattern Processor [Luc00]: 3 VLIW processing units, one control processor, 133 MHz clock rate, 1.1 GB/s DRAM bandwidth, 5 Gb/s link rate, 64 threads per processor.
- MMC nP3400 [MMC00]: 2 processing units, 220 MHz clock rate, 0.5 GB/s DRAM bandwidth, 5 Gb/s aggregate throughput, 8 threads per processor.
- Motorola C-5 [C-P99]: 16 processing units arranged in 4 clusters with reconfigurable data path, one control processor, 200 MHz clock rate, 1.6 GB/s DRAM bandwidth, 5 Gb/s line speed, 4 threads per processor.

- Tsqware TS704 [T.s99]: 4 processing units, 90 MHz clock rate, 0.3 Gb/s DRAM bandwidth.

- Vitesse Prism IQ2000 [Sit00]: 4 processing units, 200 MHz clock rate, 1.6 GB/s DRAM bandwidth, 6.4 Gb/s aggregate throughput, 5 threads per processor.

Most network processors are system-on-a-chip designs that combine processors, memory, and I/O on a single ASIC. They run extremely simple operating systems and the packet processing code is often hand-coded to achieve good performance using the small (4kB to 32kB) caches. Some processing engines are augmented by specialized instructions, multithreading, and zero-overhead context switching mechanisms. A detailed description of these network processors can be found in [Sha01].

The active router that we propose is different insofar that it is not geared towards processing of independent packets, but towards true general purpose processing of data streams that span many packets. This requires the ability to store per-flow state information (e.g. encryption keys or partial video frames) and make this state accessible to all packets of one flow. Also, we aim at complex applications that may modify the entire packet payload.

In terms of network processors, academic research has not yet addressed the topic sufficiently and has instead focused on single-processor "workstation routers." Only recently a few groups started looking into general purpose processing support in the data plane [HMS98]. Also many general-purpose parallel processor architectures have been adapted to the networking environment, like RAW [BTKM+02] and Imagine [KDR+01].

2.6 Summary

The performance results from the Active Network Node project show that single-processor software-based routers are not able to provide sufficient processing power for complex applications at increasing link speeds. The proposed router design is targeted to provide the required processing performance by exploiting the parallelism that can be found in networking workloads. The highly parallel network multi-processor is a system-on-a-chip that contains processors, memory, and I/O components on a single ASIC. In terms of processors, we mainly focus on simple RISC cores, because they provide the most processing power for their small size. If necessary, more complex architectures, DSPs, and specialized co-processors can be added.

A particularly important point of the architecture is its scalability. In terms of technology scalability, the network processor will provide increasing processing power with improvements in ASIC technology despite rapidly increasing link rates. Also, multiple processing chips can be chained together to increase the processing power of a port.

One key question that remains open is how to configure the application processing chip in terms of number of processors, and memory sizes. Chapter 4 presents an extensive analytic performance model, with which the optimal configuration for a given workload can be found. Since the results are workload-dependent, Chapter 3 first introduces a network processor benchmark, which we use to get a quantitative understanding of the processing requirements.

Chapter 3

Workload Characterization

In the previous chapter, we have seen that active networking applications can be quite processing-intensive. To get a better quantitative understanding of processing characteristics, this chapter discusses a benchmark for network processors and presents measurement results in terms of processing complexity, memory performance, instruction mix, and other metrics.

The need for a new benchmark arises from the fact that the workloads for network processors are significantly different from workloads of traditional workstation processors. Networking tasks are very short, focus on I/O, and are less complex than typical workstation task. While there are many established benchmarks for the workstation domain (most notably SPEC [Sta95]), there was no benchmark available for network processors in 1998, when this work was started. The benchmark that is presented here, *CommBench*, constitutes one of the first published results on network processor workloads [WF00].

The applications of the benchmark are described in detail and simulation results of workload characteristics are presented. Section 3.3 compares the results to measurements on SPEC and discusses implications for network processor architectures. The quantitative results obtained here are used in the performance model in Chapter 4. A shorter version of this chapter is published in [WF00].

3.1 CommBench Applications

A desirable property of any application in a benchmark is its representativeness of a wider class of applications in the domain of interest. CommBench applications have been chosen with this in mind. Therefore, the key focus is on the "kernels" of

Table 3.1: Applications of the CommBench Benchmark.

Name	Type	Application	Kernel
RTR	HPA	Radix tree routing	Lookup on tree data structure
FRAG	HPA	IP header fragmentation	Packet header checksum computation
DRR	HPA	Deficit round robin	Queue maintenance
TCP	HPA	TCP filtering	Pattern matching on header fields
CAST	PPA	Encryption	Encryption arithmetic
ZIP	PPA	Data compression	Compression arithmetic
REED	PPA	Reed-Solomon FEC	Redundancy coding
JPEG	PPA	JPEG Compression	DCT and Huffman coding

the applications, which are the program fragments containing the set of dynamically frequently used instructions. We determine the kernel of a program statistically by identifying the set of instructions, that constitute 99% or 90% of all instruction executions. If the kernel of an application is computationally similar to a wide class of applications, we can assume that the derived characteristics of the benchmark applications are representative for this class. For example, the tree based lookup in the RTR program is representative of many routing algorithms as well as packet classification schemes. The discrete cosine transform performed in the JPEG program is the basis of all JPEG and MPEG coding schemes.

CommBench applications have also been selected to represent typical workloads for both traditional routers (focus on header processing) and programmable routers (perform both header and stream processing). Thus, the applications can be divided into two groups: *Header-Processing Applications* (HPA) and *Payload-Processing Applications* (PPA). The list of applications is shown in Table 3.1.

3.1.1 Header-Processing Applications

The header-processing programs represent operations that are done on a per-packet basis and are mainly independent of the size and type of the packet payload. These applications involve a good deal of "random" logic, header field interrogation and processing, table lookup, and control. We have selected the public domain programs listed below which are likely to be operationally similar to proprietary programs.

- *RTR* is a Radix-Tree Routing table lookup program. Routing table lookups are important operations performed on every packet in a datagram-based network, and on every connection in a connection-based network. *RTR* is the radix-tree

routing algorithm from the public domain NetBSD distribution [Net]. There are more efficient routing approaches [SVSW98], however they are not freely available. *Kernel*: lookup operations on tree data structure.

- *FRAG* is an IP packet fragmentation application. IP packets are split into multiple fragments for which some header fields have to be adjusted and a header checksum computed. The checksum computation that dominates this application is performed as part of all IP packet application programs other than just forwarding. *Kernel*: packet header modifications and checksum computation.

- *DRR* is a Deficit Round Robin fair scheduling algorithm [SV95] that is commonly used for bandwidth scheduling on network links. The algorithm is implemented in one form or another in various switches currently available (e.g., Cisco 12000 series [Cis99]). *Kernel*: queue maintenance and packet scheduling for fair resource utilization.

- *TCP* is a TCP traffic monitoring application that is representative of the class of monitoring and management applications. We use *tcpdump*, a widely used tool, that is standard in BSD distributions and is based on the BSD packet filter [MJ93]. *Kernel*: pattern-matching on header data fields.

3.1.2 Payload Processing Applications

Payload-processing applications access and possibly modify the contents of a packet during network node processing. The applications are typically executed on a stream of packets. Note that each of these applications has an encoding and a decoding section. While each of these sections is executed separately, they are considered together as a single program unless they have significantly different performance characteristics.

- *CAST* is a program based on the CAST-128 block cipher algorithm that uses a 128 bit key to encrypt data for secure transmission [Ada97]. CAST-128 operates similarly to other block cipher algorithms used in current networks, such as IDEA [Lai92] and RC5 [Riv95], but CAST is in the public domain. *Kernel*: encryption arithmetic.

- *ZIP* is a data compression program based on the commonly used Lempel-Ziv (LZ77) algorithm [ZL77]. The implementation can achieve different levels of

data compression by varying the algorithm's computational complexity and exemplifies applications that permit tradeoffs between computational power and bandwidth. *Kernel*: data compression.

- *REED* is an implementation of the Reed-Solomon Forward Error Correction scheme that adds redundancy to data to allow recovery from transmission errors [RF89]. This is commonly used on unreliable data links which can be found in wireless networks. *Kernel*: redundancy coding.

- *JPEG* is a lossy compression algorithm [Wal91] for image data. It represents the class of media transcoding applications. *Kernel*: discrete cosine transform (DCT) and Huffmann coding.

Note that ZIP, JPEG, and REED decoding perform data dependent computations, while CAST and REED execute the same instructions independent of the actual data.

3.2 Measurements

There is a wide range of characteristics associated with any benchmark, and which of these impact performance depends on the underlying processor architecture and associated compiler. We have selected the following general areas of characterization, which are closely related to network processor performance as is discussed in Chapter 4: code and kernel sizes, computational complexity, instruction frequency, and cache performance.

3.2.1 Tools and Input Data

In collecting data, all the benchmark programs were run on SUN UltraSparc II processors operating under the SunOS 5.7. The C compiler used was *gcc* 2.8.1 (optimization level O2). The O2 level was selected because the compiler only performs optimizations that are independent of the target processor and does not exploit particular architectural features (e.g., loop unrolling for superscalar machines). To determine the influence of the compiler, for selected statistics the *gcc* to the *cc* 4.2 compiler were compared. Differences in the generated instruction mix were limited to 1-2% for each instruction class and cache performance of the generated code was also very similar.

Table 3.2: Code Size for CommBench Applications.

CommBench Program	Type	Code Size C lines	Code Size Object bytes
TCP	HPA	19,100	352,000
JPEG	PPA	18,300	260,000
ZIP	PPA	6,500	117,000
RTR	HPA	1,130	16,000
REED	PPA	410	6,900
CAST	PPA	350	19,500
DRR	HPA	100	2,500
FRAG	HPA	100	2,400
Average		5,750	97,000

For run time instruction mix analysis, Shade [CK94] and SpixTools [Cme93] were used. These tools simulate and analyze programs on a Sparc processor. For the cache simulations, Dinero [EH98], a uniprocessor cache simulator, was used.

The benchmark programs were executed with a variety of input data to see the effect on program operation characteristics. While the header-processing applications require data inputs in a particular format for each program (i.e., TCP requires raw packet header, while RTR lookups requires IP addresses), the payload-processing applications, except for JPEG, can process any data stream. For these applications we measured instruction mix and cache behavior for HTML data (plain text), binary program code, and JPEG coded image data. While CAST and REED perform identically on any data, ZIP shows differences on data that has already been entropy encoded (i.e., JPEG data). To account for this variation, the input for the benchmark measurements was chosen with an equal mix of all three data types.

3.2.2 Code and Computational Kernel Sizes

One can view the size of an application along a number of different dimensions ranging from source code size to the number of bytes most often referenced during execution. Both static (i.e., lines of C code, bytes of compiled code) and dynamic code size information (i.e., instructions executed at least once, accounting for 90% and 99% of execution) was collected.

The size of the source code and compiled code of each program in CommBench is shown in Table 3.2. The object size data does not include the large but little used

Table 3.3: CommBench Dynamic Kernel Size.

CommBench Program	Instructions at least once	Instructions for 99%	Instructions for 90%
TCP	7,257	317	232
JPEG	6,155	804	504
RTR	3,805	1,371	387
ZIP	3,538	555	296
CAST	2,529	716	642
REED	1,510	48	23
DRR	1,353	70	36
FRAG	1,258	97	80
Average	3,430	500	275

dynamically linked libraries (up to 300 kilobytes on the SUN Solaris system). CommBench programs are between 2.4kB and 350kB in size. The variation in CommBench code size stems from the different environments in which the applications have been developed. DRR and FRAG are non-commercial proof-of-concept implementations, while ZIP and JPEG are industrial strength implementations with a multitude of options. This has an impact on static code analysis, but dynamic run-time analysis indicates that all applications execute within a fairly small kernel.

The dynamic kernel characteristics of CommBench programs are shown in Table 3.3. The first column indicates the number of instructions which have been executed at least once. Note that the average is 3,430 instructions (13,720 bytes), which is significantly less than the average unlinked object code size (97,000 bytes). Even when one removes from the object code size the roughly 15% which corresponds to data fields, this indicates the presence of a significant amount of code that is never executed. This 'dead' code typically corresponds to code for error handling conditions or rarely used input data formats. This observation is reinforced by examining the number of instructions which constitute 90% or 99% of the instructions executed. With 23 to 640 instructions comprising 90% of the workload of a program, it can be seen that network processor applications are quite small in size.

A common notion used in processor design is the "90/10 rule;" that is 90% of executed instructions are derived from 10% of the instructions in the program. Figure 3.1 is a visual representation of this rule showing kernel size in relation to the total number of instructions. A steeply rising curve indicates that only a few instructions are responsible for most of the runtime computation. Only instructions

that are executed at least once are considered. In CommBench, RTR, ZIP, and JPEG have kernels that follow the 90/10 rule very closely with others having smaller kernels (e.g., FRAG, DRR, REED follow 95/5), and in a couple of cases larger kernels (e.g., TCP follows 85/15). CAST has a basically linear behavior due to a fairly large inner loop that repeats many instructions the same number of times.

3.2.3 Computational Complexity

In networking, streaming data through the programs is a significant portion of an application. In contrast, in most traditional workstation benchmarks, the input is limited to a few parameters and little input data and the output is only a few result values. Given clearly defined I/O data streams for each CommBench application, it is possible to define the computationally complexity of each application with respect to the number and size of the processed packets. This complexity measure helps in determining certain aspects of performance as a function of expected workload.

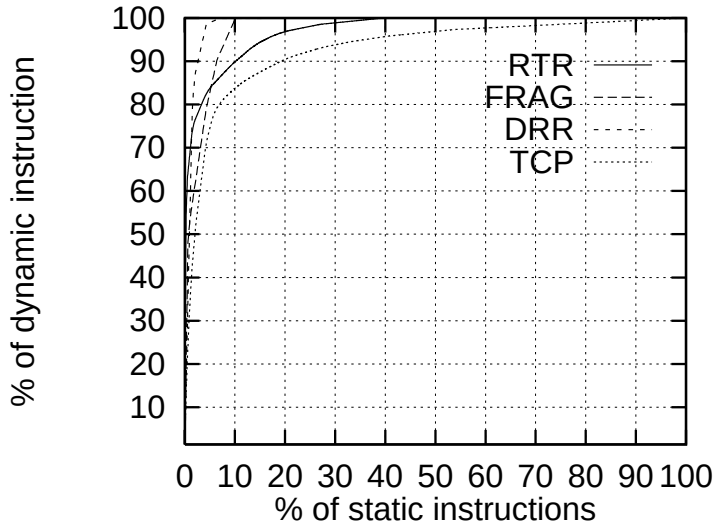
We define the computational complexity $compl_{a,l}$ to be the number of instructions per byte required for application a operating on a packet of length l:

$$compl_{a,l} = \frac{instructions\ executed\ by\ a}{l}. \tag{3.1}$$

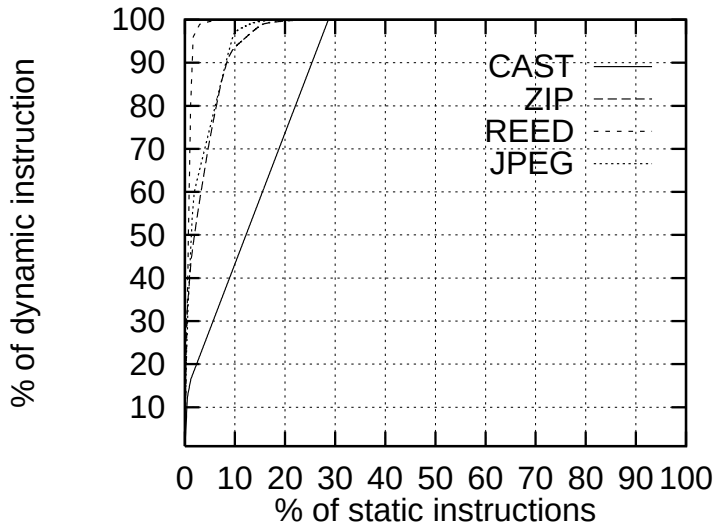
The computational complexity does not reflect memory system performance since it is based on the number of instructions rather than the number of cycles executed. For header processing, l is taken to be 64, 576 and 1536 bytes (i.e., minimum IP-packet size, minimum MTU (maximum transfer unit) over IP, and maximum Ethernet packet size). The minimum $l = 64$ bytes is also in the range of ATM cell size (53 bytes). For payload processing applications l is effectively equal to infinity. That is, we consider data streams of sufficient length ($l \geq$ 1MB) so that startup overhead processing is negligible. Table 3.4 shows the complexity of CommBench applications. Section 3.4 discusses how this complexity measure is used to determine the required processing power for a given application and link rate.

3.2.4 Instruction Set Characteristics

The instruction mix gives an indication on the type of instructions executed in the benchmark. Averages for CommBench, and its two components HPA and PPA are given in Figure 3.2. Table 3.5 presents this same data sorted by frequency.



(a) Header-Processing Applications



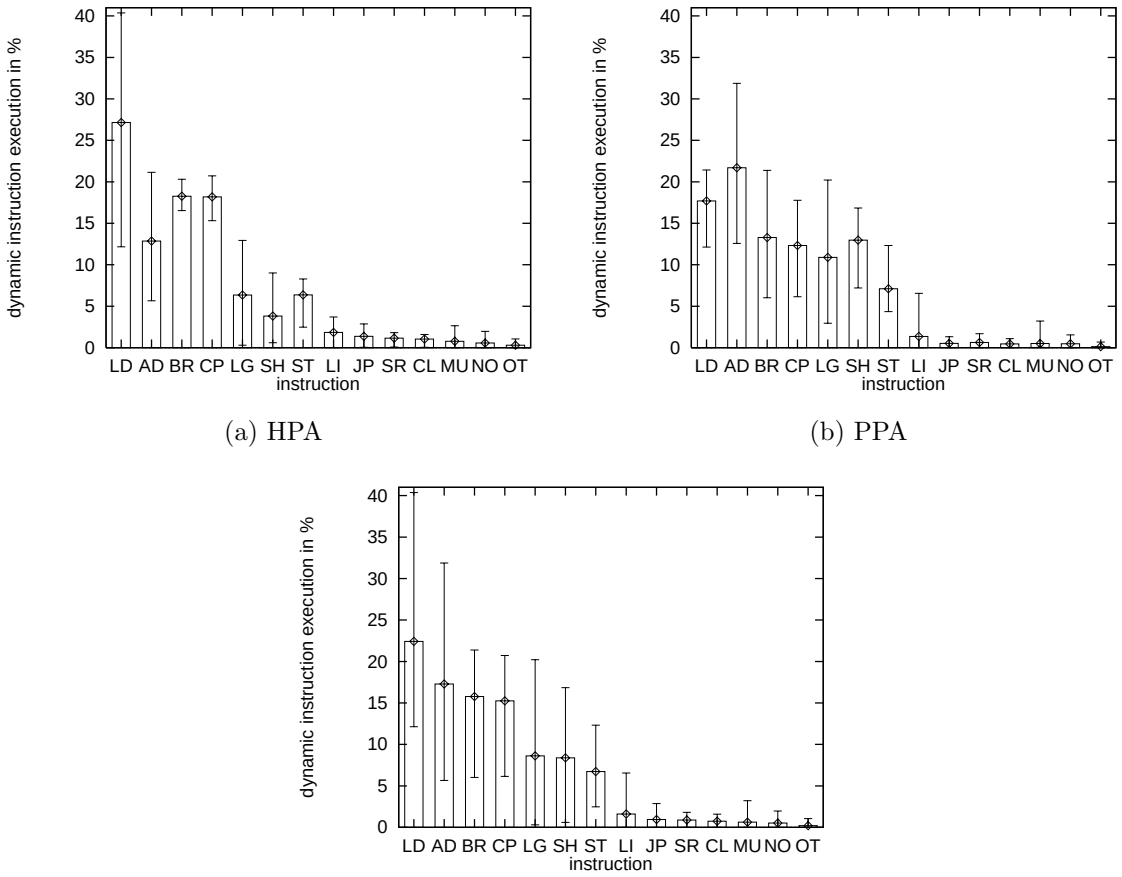
(b) Payload-Processing Applications

Figure 3.1: Locality in CommBench Applications.

Table 3.4: Computational Complexity of CommBench Applications. The results are given in instructions per byte.

HPA a	$compl_{a,64}$	$compl_{a,1536}$
TCP	10.3	.4
FRAG	7.7	.3
DRR	4.1	.2
RTR	2.1	.1

PPA a	$compl_{a,\infty}$ (enc)	$compl_{a,\infty}$ (dec)
REED	603	1052
ZIP	226	35
CAST	104	104
JPEG	81	60



(a) HPA

(b) PPA

(c) CommBench

Figure 3.2: CommBench Instruction Frequencies. Error bars indicate the minimum and maximum of instruction frequencies encountered for any single application (LD = load, AD = add/sub, BR = conditional branch, CP = compare, LG = logic, SH = shift, ST = store, LI = load immediate, JP = jump and link, SR = save/restore, CL = call, MU = mult, NO = nop, OT = other).

Table 3.5: Ordered Instruction Frequencies for CommBench.

CommBench Average	%	CommBench HPA	%	CommBench PPA	%
load	22	load	27	add/sub	22
add/sub	17	cond. branch	18	load	18
cond. branch	16	compare	18	cond. branch	13
compare	15	add/sub	13	shift	13
logic	9	store	6	compare	12
shift	8	logic	6	logic	11
store	7	shift	4	store	7
load imm.	2	load imm.	2	load imm.	1
jmpl	1	jmpl	1	save/restore	1

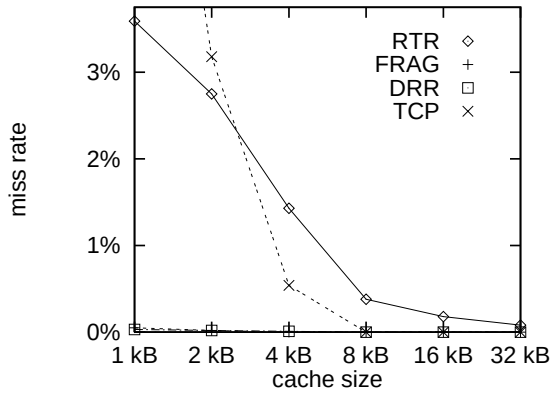
The two components of CommBench, HPA and PPA, show significant differences in the instruction execution frequencies. The usage of load and add/sub differs by about 10% and 5% for branch, compare, and logic. This indicates that Header-Processing Applications are more dominated by "random logic" that heavily uses load, branch, and compare. Payload-Processing Applications are more dominated by add, shift, and logic, which indicates their emphasis on computationally intense tasks.

3.2.5 Memory Hierarchy Characteristics

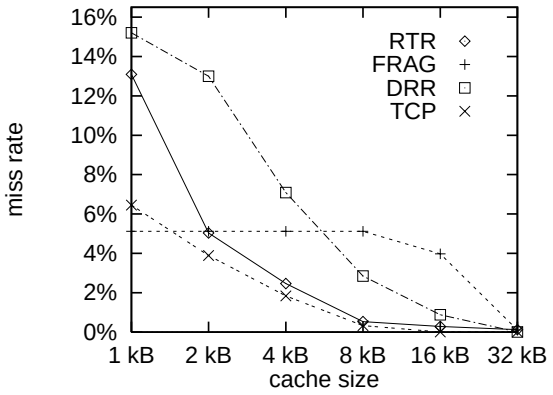
An important part of any processor design is its memory hierarchy. Figure 3.3 shows the results for a 2-way associative instruction and data cache from 1KB to 32KB. Other caches with different associativity were also investigated. For the direct mapped cache the rule of thumb holds which states that the miss rate is about 1.5 to 2 times that of a 2-way cache. The differences between 2-way, 4-way, and 8-way associative caches are minor, hence the gain for going to higher associativity given the additional chip area costs are limited.

The CommBench HPA and PPA cache performance is shown in Figure 3.4. The following points are relevant:

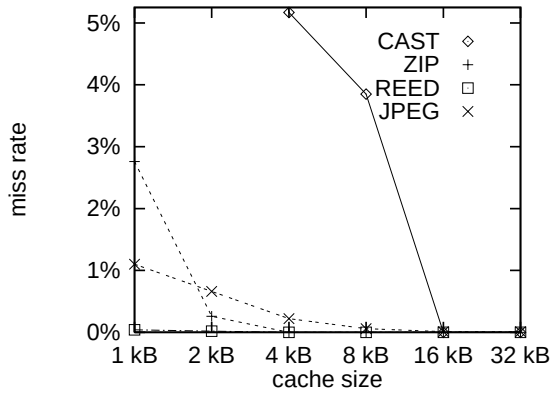
- Due to the relatively small CommBench program kernels, CommBench instruction miss rates are very low. For an 8KB instruction cache, miss rates under 0.5% can be achieved for all but the CAST program. For 16KB instruction caches, all applications achieve miss rates below 0.2%.



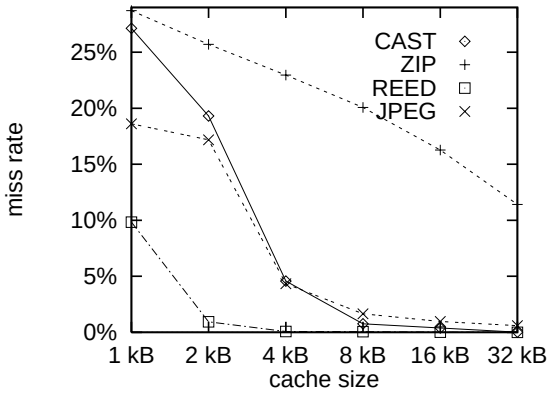
(a) HPA Instruction Cache.



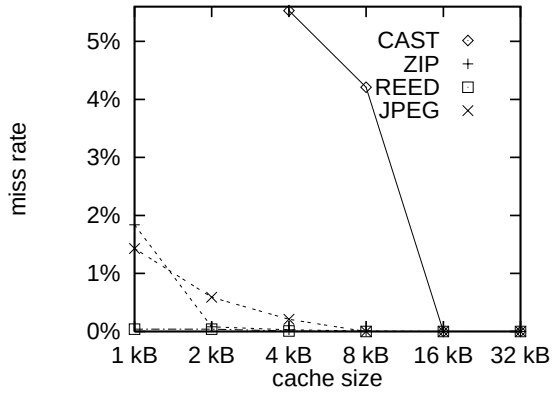
(b) HPA Data Cache.



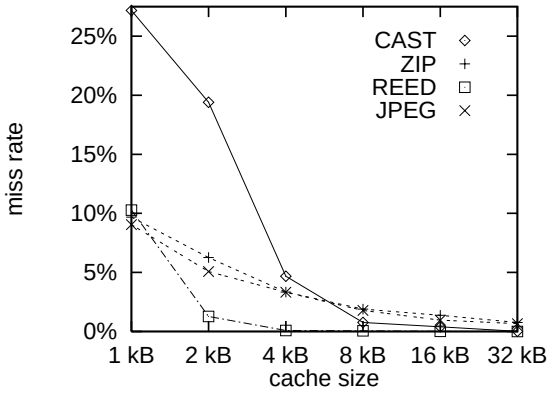
(c) PPA Instruction Cache (Encoding).



(d) PPA Data Cache (Encoding).

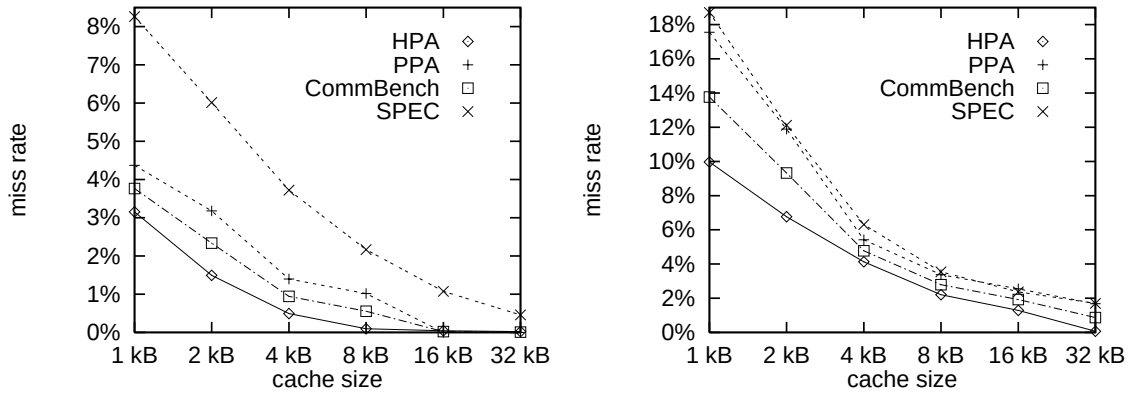


(e) PPA Instruction Cache (Decoding)



(f) PPA Data Cache (Decoding)

Figure 3.3: Instruction Cache and Data Cache Miss Rates for CommBench Applications.



(a) Instruction Cache Average.

(b) Data Cache Average.

Figure 3.4: Average Instruction Cache and Data Cache Miss Rates for CommBench, HPA, and PPA.

Table 3.6: Average Ratio of Miss Rate at Smaller Line Size to Miss Rate at the Larger Line Size.

Cache Type	16 vs. 32 byte	32 vs. 64 byte
Instruction	1.67	1.18
Data	1.36	1.26

- Data cache miss rates are below 1% for a 16KB cache, except for the ZIP and FRAG applications.

- For payload-processing applications, encoding and decoding programs have almost identical instruction and data cache performance. Only ZIP has a much higher data cache miss rate for the encoding application.

The instruction miss rates for CommBench also vary as a function of the cache line size. Table 3.6 shows that average miss rates decrease with increasing line size. The step from 16 to 32 bytes decreases the miss rates by 1.67 for instruction cache and 1.36 for data cache. Going to 64 bytes decreases the miss rate further, but only by 1.18 for instruction cache and 1.26 for data cache. There are two applications (CAST and RTR) that have higher miss rates with a 64 byte line size than with a 32 byte line size. The overall performance effect of using increased line size depends not only on the miss rate, but also on the miss penalty. This is processor implementation dependent, though, and is therefore not considered here.

Table 3.7: SPEC Code Size.

SPEC Program	Code Size C lines	Code Size Object bytes
126.gcc	206,000	1,950,000
147.vortex	67,200	1,150,000
132.ijpeg	31,200	594,000
099.go	29,200	558,000
134.perl	26,900	544,000
124.m88ksim	19,900	404,000
130.li	7,600	139,000
129.compress	1,930	81,700
Average	48,700	678,000
CommBench Avg.	5,750	97,000

3.2.6 Summary of Characteristics

The dynamic code sizes of CommBench applications are only in the order of a few hundred to a thousand instructions, which characterizes the simple and focused applications in this domain. The instruction frequencies of Header-Processing Applications are dominated by load, compare, and branch. Payload-Processing Applications execute significantly more add/sub, shift, and logic operations. With respect to the cache performance, CommBench applications have very low miss rates, which are due to the small kernel sizes of CommBench applications. For an instruction cache of 16 Kbytes the miss rates drop below 0.1%. For data cache performance, payload processing applications have slightly higher miss rates than header processing applications.

3.3 Comparison to SPEC

To highlight the differences between workloads for network processors and those for workstations, we briefly compare the CommBench results to measurements on SPEC [Sta95]. The SPEC results were obtained using the identically same simulation environment as for CommBench. Table 3.7 shows the code size for SPEC applications. Comparing these results to the average for CommBench (from Table 3.2), it can be observed that SPEC applications are about one order of magnitude larger than CommBench applications. A similar observation can be made for the dynamic code size shown in Figure 3.8.

Table 3.8: SPEC Dynamic Kernel Characteristics.

SPEC Program	Instructions at least once	Instr. for 99%	Instr. for 90%
126.gcc	124,246	43,983	15,899
147.vortex	60,630	10,136	1,715
099.go	53,629	17,511	6,530
132.ijpeg	12,627	1,735	949
124.m88ksim	12,313	2,154	875
134.perl	12,284	683	542
130.li	7,341	990	408
129.compress	2,842	352	227
Average	35,700	9,700	3,390
CommBench Avg.	3,430	500	275

With respect to the instruction frequency, SPEC displays some similarity to CommBench. The error-bar for add shows that for certain applications, SPEC is quite similar to PPA in CommBench, even through the average is closer to HPA.

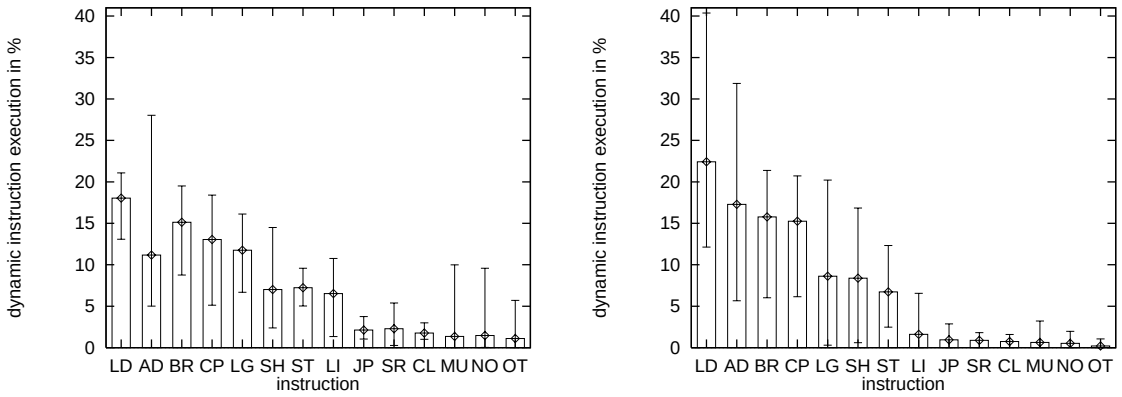
The differences in cache performance can be seen in Figure 3.4. The instruction cache miss rate for SPEC is much higher than CommBench due to the larger program sizes. These results confirm the assumption that network processors can perform well with smaller caches than workstation processors.

3.4 Architectural Implications

There are several implications for the programmable router design that can be derived from the above results. For one, the definition of computational complexity gives the opportunity to estimate the required processing power for a given scenario. Assuming a link speed of OC-24 (1.244 Gbps), 64-byte packets, and processing, which consists of routing (RTR) and scheduling (DRR), we can compute the necessary processing power in million instructions per second (MIPS):

$$\begin{aligned} processing\ power &= link\ rate \cdot \textstyle\sum_{a=\{RTR,DRR\}} compl_{a,64} = \\ &= 1.244\ Gbps \cdot \tfrac{1byte}{8bit} \cdot (2.1 + 4.1) = 964\ MIPS. \end{aligned} \tag{3.2}$$

Current processors are able to process around 1000 MIPS, which indicates that simple forwarding could be performed by a single-processor software-based router. However



(a) SPEC (b) CommBench

Figure 3.5: SPEC and CommBench Instruction Frequencies. Error bars indicate the minimum and maximum of instruction frequencies encountered for any single application. (see Figure 3.2 for x-axis legend).

the requirements for payload processing (e.g., CAST) for a gigabit link are beyond current single-processor capabilities:

$$\begin{aligned} processing\ power &= link\ rate \cdot \textstyle\sum_{a=\{CAST\}} compl_{a,\infty} = \\ &= 1.244\ Gbps \cdot \tfrac{1byte}{8bit} \cdot 104 = 16,200\ MIPS. \end{aligned} \tag{3.3}$$

Such processing requirements can only achieved by multiprocessor systems as proposed for the programmable router design in Chapter 2.

From the differences in instruction mix, we can conclude that it might be beneficial to have a few specialized processors on the APC. Applications can make use of the special processor should be processed there. One example is a hybrid configuration of RISC processors and DSPs for media applications.

Finally, the results on the memory performance of CommBench indicate that smaller caches than they are currently used in workstation processors are sufficient to achieve efficient execution. In Chapter 4, we will see that caches in the order of 16-32kB are a good design choice.

3.5 Related Work

There is a long history of developing synthetic benchmarks (Whetstone [CW76], Dhrystone [Wei84]) and realistic benchmarks (SPEC [Sta95]). The most popular benchmark associated with workstations have been the SPEC suites. Benchmarks aimed towards other application classes have also been successfully developed with two examples being TPC [Tra98], for transaction processing applications, and SPLASH [WOT+95], for scientific applications executing on parallel processors.

In addition to differences in program execution characteristics, traditional benchmarks do not reflect other aspects of the telecommunications environment. One significant shortcoming is the missing focus on clearly defined I/O. A recent benchmark, that addresses I/O issues in the context of multimedia applications is the MediaBench benchmark [LPMS97] which consists of programs implementing various compression and coding algorithms for streaming voice, audio, and video data (e.g., JPEG, MPEG, GSM, etc.). However, multimedia transcoding is only one part of the network processor applications domain. Additionally, such processors must perform a wide variety of logical control operations not significantly present in MediaBench. CommBench includes streaming data flow based applications similar to those found in MediaBench, and additional packet-based processing tasks such as routing and data forwarding.

After the development of CommBench, several other network processor benchmarks have been published. An industrial attempt to benchmarking network processors is done by the Embedded Microprocessor Benchmarking Consortium (EEMBC) [EEM]. EEMBC maintains a set of benchmarks for embedded systems, which also include a few simple networking applications, however, the focus in mainly on header-processing applications. Crowley *et al.* have defined a set of networking benchmark applications [CFBB99] in the context of their research. There is some overlap with CommBench, but the aim of the analysis is more towards opportunity for exploiting ILP than characterizing RISC processor performance. Recently, NetBench has been developed by Memik *et al.*, which distinguished between different levels of operation [MMSH01]. Micro-level, IP-level, and application-level programs are defined. PPA in CommBench are somewhat similar to applications-level programs in NetBench. HPA correspond to micro-level and IP-level.

There are still several open questions on benchmarking network processors. In particular the lack of a clearly defined architecture for network processors hinders the

efforts to develop a single standard benchmark similar to SPEC in the workstation domain. There are some attempts to structure benchmarking methodologies to accommodate different network processor architectures [CYB+02], [SBM02], but more progress in this area can be expected over the next few years.

3.6 Summary

CommBench provides a quantitative understanding of the processing requirements and characteristics of the networking domain. In particular, the definition of computational complexity is a useful metric for estimating processing requirements. The measurements of code size, instruction mix, and cache performance is used with the performance model in the following chapter to determine APC configurations that are optimized for particular workloads.

Chapter 4

Performance Model

One major difficulty in the area of network processor design is the lack of a methodology for comparing the performance of different designs. Unlike workstation processors, where the basic design is similar in most machines, network processors range from straightforward RISC multiprocessors to complex combinations of processors, specialized co-processors, and custom logic. Qualitative comparisons are only of limited use as there are very basic differences among different network processor architectures. Therefore, we focus on a quantitative evaluation of this domain.

In this chapter, a general performance model for a basic network processor design is developed. Due to the generality of the network processor model, it is applicable to a range of network processor architectures. The evaluation requires a set of easily obtainable technology and application parameters. The presented results show tradeoffs for various design alternatives and are used to derive the optimal configuration for the Application Processing Chip in the programmable router. This chapter is published in [WF02] and [FW02].

4.1 Analytic Model

The single-chip network processor architecture used in the analytic model is shown in Figure 4.1. It is a generalization of the APC design discussed in Chapter 2. The system contains a number of identical multithreaded general-purpose RISC processors with t threads. Each processor has its own instruction and data cache of size c_i and c_d. Groups of n processors are clustered together and share a memory interface for off-chip memory accesses. The network processor consists of m clusters. By varying the number of processors, threads, clusters, and sizes of caches, a broad range of

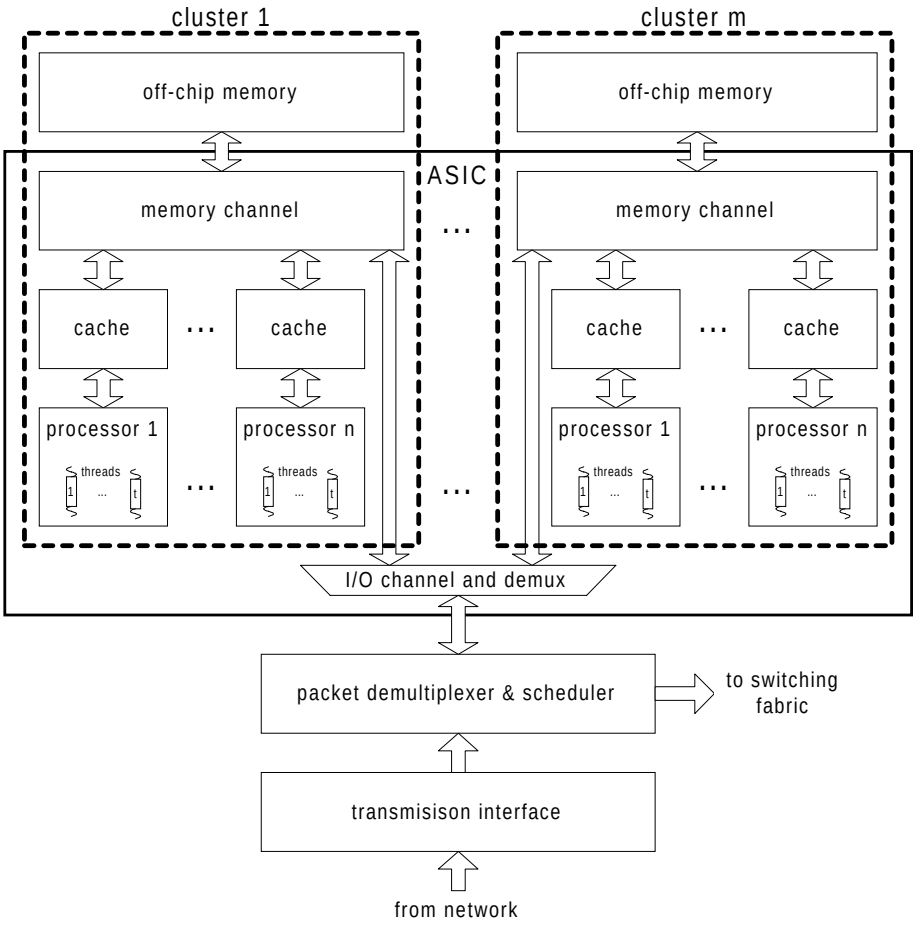


Figure 4.1: Network Processor Architecture for Performance Model. This system is an abstraction from the architecture in Figure 2.4.

processor architectures can be considered. The various system parameters used in the performance model are listed in Table 4.1.

The goal of the model is to find the "optimal" configuration of a network processor for a given workload. Optimal, in this context, means obtaining the maximum processing power per unit of chip area. We develop analytic expressions for the processing power, IPS, (Instructions Per Second) and the area, $area$, associated with a given architecture configuration (e.g., number of processors, sizes of caches, etc.). From these expressions we can obtain $IPS/area$ and find its maximum as a function of the various configuration parameters, thus developing an "optimal" architecture. In the remainder of this section, we discuss how to obtain IPS and $area$ in terms of system and workload characteristics.

Table 4.1: System Parameters for Performance Model.

Component	Symbol	Description
processor	clk_p	processor clock frequency
	t	number of simultaneous threads on processor
	ρ_p	processor utilization
program a	f_{load_a}	frequency of load instructions
	f_{store_a}	frequency of store instructions
	$mi_{c,a}$	i-cache miss probability for cache size c_i
	$md_{c,a}$	d-cache miss probability for cache size c_d
	$dirty_{c,a}$	prob. of dirty bit set in d-cache of size c_d
	$compl_a$	complexity (instr. per byte of packet)
caches	c_i	instruction cache size
	c_d	data cache size
	$linesize$	cache line size of i- and d-cache
off-chip memory	τ_{DRAM}	access time of off-chip memory
memory channel	$width_{mchl}$	width of memory channel
	clk_{mchl}	memory channel clock frequency
	ρ_{mchl}	load on memory channel
I/O channel	$width_{io}$	width of I/O channel
	clk_{io}	clock rate of I/O channel
	ρ_{io}	load on I/O channel
cluster	n	number of processors per cluster
ASIC	m	number of clusters and memory channels
	$s(x)$	actual size of component x, with $x \in \{ASIC, p, c_i, c_d, io, mchl\}$

4.1.1 Processing Performance

The processors are typical RISC processors, which ideally execute one instruction per cycle. Processor stalls can occur when the on-chip cache cannot satisfy a memory request. (We assume that the on-chip SRAM cache can be accessed in a single cycle.) In such a case, the processor switches to the next thread to hide the memory access latency. We assume that context-switching is done in hardware with zero cycle overhead. This means that if one thread stalls, another thread can immediately start processing with no cycle delay.

For a single processor, processing power can then be expressed as the product of the processor's utilization, ρ_p, and its clock rate, clk_p. The processing power of the entire NP can be expressed as the sum of processing power of all the processors on the chip. Thus, with m clusters of processors and n processors per cluster:

$$IPS = \sum_{j=1}^{m} \sum_{k=1}^{n} \cdot \rho_{p_{j,k}} \cdot clk_{p_{j,k}}. \tag{4.1}$$

If all processors are identical and run the same workload, then on average the processing power is:

$$IPS = m \cdot n \cdot \rho_p \cdot clk_p. \tag{4.2}$$

A key question is how to determine the utilization of the processors. In the extreme case where there is a large number of threads per processor, large caches that reduce memory misses, and low memory miss penalties, the utilization approaches 1. However, a large number of thread contexts and larger caches require more chip area, reducing the number of processors.

4.1.2 Chip Area

The on-chip area equation for an NP configuration in our general architecture is:

$$area_{NP} = s(io) + \sum_{j=1}^{m} (s(mchl) + \sum_{k=1}^{n} (s(p_{j,k}, t) + s(c_{i_{j,k}}) + s(c_{d_{j,k}}))). \tag{4.3}$$

This is the summation over all the system component areas shown in Figure 4.1. With identical processor configurations, this can be simplified to:

$$area_{NP} = s(io) + m \cdot (s(mchl) + n \cdot (s(p,t) + s(c_i) + s(c_d))). \tag{4.4}$$

The processor size, $s(p,t)$, depends on the number of hardware threads and is therefore expressed as $s(p,t)$, a function of t. We model the processor size in terms of two components. The first component, size $s(p_{basis})$, is independent of the number of supported threads. It represents the basic processor logic (e.g., ALU, pipeline control, branch prediction, etc.). The second component, size $s(p_{thread})$, relates to logic associated with a thread (e.g., thread context registers, associated logic, etc.). This thread component is modelled as increasing linearly with the number of threads, t. While this might be optimistic for large numbers of threads, it is a reasonable assumption for the relatively small number of threads considered here. Thus, the processor size is:

$$s(p,t) = s(p_{basis}) + t \cdot s(p_{thread}). \tag{4.5}$$

The size of a memory or I/O bus also consists of a basis area plus the on-chip area of the pin drivers and pads. The total size depends on the width of the bus:

$$s(mchl) = s(mchl_{basis}) + width_{mchl} \cdot s(mchl_{pin}). \tag{4.6}$$

The number of pins depends on the bus clock, clk_{mchl}, and the required bus bandwidth, bw_{mchl}. Thus:

$$s(mchl) = s(mchl_{basis}) + \left\lceil \frac{bw_{mchl}}{clk_{mchl}} \right\rceil \cdot s(mchl_{pin}). \tag{4.7}$$

with the equivalent equation being used for the I/O channel.

Equation 4.4 and the subsequent Equations 4.5-4.7 define the area requirements of system configurations. Before this can be used in the evaluation of the overall performance metric $IPS/area_{NP}$, the processor utilization must be determined so that IPS from Equation 4.2 can be evaluated. In particular, the processor utilization, ρ_p, depends on the performance of the memory system.

4.1.3 Memory System

The performance of the network processor is determined by the utilization of the individual processing engines. A RISC processor is fully utilized as long as memory

misses do not cause a processor stall. Other stalls due to hazards, such as branch misprediction, are not considered here since, with modern processor and compiler designs, they generally have a relatively small effect compared to the effects of cache misses. Using the model proposed and verified by Agarwal [Aga92], the utilization $\rho_p(t)$ of a multithreaded processor is given as a function of the cache miss rate p_{miss}, the off-chip memory access time τ_{mem}, and the number of threads t as:

$$\rho_p(t) = 1 - \frac{1}{\sum_{i=0}^{t} \left(\frac{1}{\tau_{mem} \cdot p_{miss}}\right)^i \frac{t!}{(t-i)!}}. \tag{4.8}$$

To illustrate the overall trend in this equation, we can simplify Equation 4.8 by ignoring the second and higher order terms of the summation (i.e., $i = 0, 1$). Thus:

$$\rho_p(t) = 1 - \frac{1}{1 + \frac{t}{\tau_{mem} \cdot p_{miss}}} = \ldots = \frac{t}{(t + \tau_{mem} \cdot p_{miss})}. \tag{4.9}$$

Note from this expression that, as expected, the utilization decreases with increasing miss rates and with increasing miss penalties for off-chip memory accesses. The larger the number of threads, t, the less the impact of τ_{mem} and p_{miss}, since more threads are available for processing and processor stalls are less likely. In the limit $\lim_{t\to\infty} \rho_p(t) = 1$. While it is desirable to run processors at high utilization there is an increasing area cost with this as indicated in Equation 4.5. This impacts overall performance since the added processor area due to more thread contexts leads to less area available for caches and thus can lead to higher miss rates. The analysis of this tradeoff requires expressions for the memory access time, τ_{mem}, and the cache miss rate, p_{miss}.

Off-Chip Memory Access

We assume the memory channel implements a FIFO service order on the memory requests in such a way that they can be interleaved in a split transaction fashion. The total off-chip memory request time, τ_{mem}, thus has three components: the bus access time, τ_Q, the physical memory access time, τ_{DRAM}, and the cache line transmission time, $\tau_{transmit}$ (all represented in terms of numbers of processor clock cycles):

$$\tau_{mem} = \tau_Q + \tau_{DRAM} + \tau_{transmit}. \tag{4.10}$$

The DRAM access time is a fixed technology parameter and the cache line transmission time can be determined from the memory channel width, clock rate, and cache line size. The queuing time, however, depends on the load on the memory channel, which depends on the number of processors that share the memory channel, the number of threads per processor, and the cache miss rates. This system component can be simply modelled as a single server queuing system with n processors that generate requests. The request distribution can be modelled as geometrically distributed random variables (as suggested in [Aga92]). Based on the average cache miss rate of a thread (see Equation 4.14 below), the parameter of the geometric random variable is p_{miss}. The number of requests per processor is limited to t, which corresponds to the situation where all the processor threads are stalled and the processor is idle until a memory request is served. The service time for the memory channel is taken to be deterministic with parameter $1/\tau_{transmit}$.

This model can be slightly modified to make it more suitable for the analytical evaluation. Instead of considering n processor sources each providing up to t requests, we model the system as a single finite source having up to $n \cdot t$ requests. Since each of the n sources generates requests at a mean rate p_{miss}, the single source model generates requests at a rate $n \cdot p_{miss}$.

Assuming an exponential distribution rather than a geometric and ignoring the limit of $n \cdot t$ customers, the queuing system can be approximated by a M/D/1 queuing system. The request rate is $\lambda = n \cdot p_{miss}$ and the deterministic service rate is $\mu = 1/\tau_{transmit}$.

The M/D/1 model is a reasonable approximation to the real system, which has a finite source population. Figure 4.2 shows the average queue length for the simulated real finite source system and the analytic result for the M/D/1 system. The number of threads in this example is $t = 8$, the number of processors is $n = 4$, and the service time $\tau_{transmit} = 40$. The M/D/1 model has no constraint on the maximum number of requests and therefore reaches a much larger queue length for very high loads (i.e., $> 90\%$). For a more typical load of $\rho_{mchl} = 0.5 \ldots 0.9$, the difference between M/D/1 and the other models is relatively small. Furthermore, below 50% load the queue length is small enough for both models to have relatively little effect on the overall performance. Therefore, we will use the M/D/1 model for an approximation of the queuing time τ_Q.

The bus access time, τ_Q, is then given by the queuing time of the M/D/1 system, which is

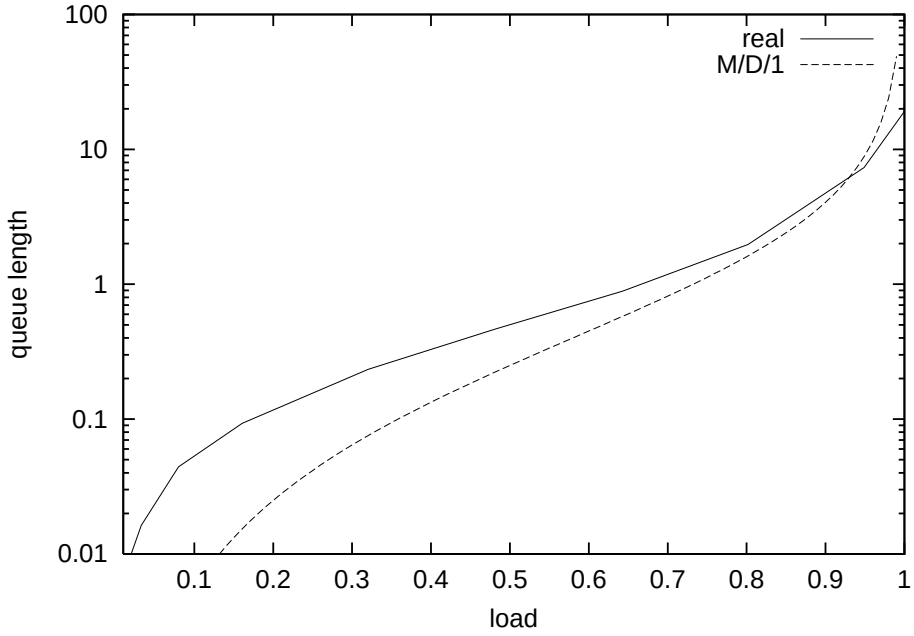


Figure 4.2: Comparison of Average Memory Queue Length for Different Queuing Models.

$$\tau_Q = \frac{\rho_{mchl}^2}{2(1-\rho_{mchl})} \cdot \frac{linesize}{width_{mchl}} \cdot \frac{clk_p}{clk_{mchl}}. \tag{4.11}$$

With a fixed DRAM access time, τ_{DRAM}, and a transmission time of

$$\tau_{transmit} = \frac{linesize}{width_{mchl}} \cdot \frac{clk_p}{clk_{mchl}}, \tag{4.12}$$

we can substitute in Equation 4.10 to obtain the memory access time:

$$\tau_{mem} = \tau_{DRAM} + \left(1 + \frac{\rho_{mchl}^2}{2(1-\rho_{mchl})}\right) \cdot \frac{linesize}{width_{mchl}} \cdot \frac{clk_p}{clk_{mchl}}. \tag{4.13}$$

On-Chip Cache

The remaining component needed to evaluate the utilization expression (Equation 4.8) is the cache miss rate p_{miss}. For a simple RISC style load-store processor running application a, the miss probability is given as [HP95]:

$$p_{miss,a} = mi_{c,a} + (f_{load_a} + f_{store_a}) \cdot md_{c,a}, \tag{4.14}$$

where $mi_{c,a}$ and $md_{c,a}$ are the instruction and data cache miss rates, and f_{load_a} and f_{store_a} are the frequency of occurrence of load and store instructions when executing application a. The instruction and data cache miss rates depend on the application, the cache sizes, and the effects of cache pollution due to multi-threading.

Cache pollution from multi-threading reduces the effective cache size that is available to each thread. On every memory stall, a thread requests one new cache line (replacing the least recently used line). While the thread is stalled, $t-1$ other threads replace one line. In steady-state, each thread can use $\frac{1}{t}$ of the available cache. If the working set size of a thread is very small, its effective cache usage could be less than $\frac{1}{t}$ (and the other threads use slightly more). In a network processor, we expect the cache sizes to be smaller than the working set size due to chip area constraints, which leads to equal sharing of the available cache between threads. Thus, the effective cache size that is available to a thread is:

$$c_{i,eff} = \frac{c_i}{t}, \quad c_{d,eff} = \frac{c_d}{t}. \tag{4.15}$$

The miss rates used in Equation 4.14 refer to this effective cache size.

4.1.4 Memory and I/O Channels

The expression for miss rate, p_{miss}, (Equation 4.14) and for total memory access time, τ_{mem}, (Equation 4.10) can now be substituted into Equation 4.8 to obtain processor utilization. In order to do this, we need to fix the memory channel load, ρ_{mchl}, because τ_Q depends on ρ_{mchl}. With the memory channel load given, we can then determine the utilization of a single processor. With the utilization given, the memory bandwidth, $bw_{mchl,1}$, required by a single processor is:

$$bw_{mchl,1} = \rho_p \cdot clk_p \cdot linesize \cdot (mi_c + (f_{load} + f_{store}) \cdot md_c \cdot (1 + dirty_c)). \tag{4.16}$$

In this equation, we have to consider the case where a dirty cache line needs to be written back to memory. The probability of the dirty bit being set on a cache line is $dirty_c$. In Equation 4.14, considering dirty cache lines was not necessary, since a write-back does not stall the processor. In practice, the write-back only increases the required memory bandwidth slightly and Equation 4.16 can be approximated by

$$bw^{*}_{mchl,1} = \rho_p \cdot clk_p \cdot linesize \cdot p_{miss}. \tag{4.17}$$

In the results below, the dirty bit is considered. The number of processors, n, in a cluster is then the number of processors that can share the memory channel without exceeding the specified load

$$n = \left\lfloor \frac{width_{mchl} \cdot clk_{mchl} \cdot \rho_{mchl}}{bw_{mchl,1}} \right\rfloor. \tag{4.18}$$

This gives us a complete cluster configuration for all ranges of cache sizes and thread contexts. Finally, we need to determine the bandwidth that is required for the I/O channel. The I/O channel is used to send packets to the processing engines and back out. Thus, each packet traverses the I/O channel twice. From Equation 3.1, we get a relation between the number of instructions executed in processing a packet and the size of the packet. The I/O channel is operated at a load of ρ_{IO}; thus, the I/O channel bandwidth for the entire network processor is:

$$bw_{IO} = 2 \cdot \frac{IPS}{compl \cdot \rho_{IO}}. \tag{4.19}$$

Finally, the network processor is limited in the number of pins that the package can have. As a rough estimate, we add the number of pins required by the I/O and memory channels, which depends on their respective width, to the control pins for the network processor:

$$pins_{NP} = pins_{IO} + m \cdot pins_{mchl} + pins_{control}. \tag{4.20}$$

We can see below that, for our basic architecture, the number of pins that can be supported do not pose a practical limit on the network processor.

4.1.5 Optimization

With the performance and area of the network processor expressed in terms of cache configurations, application characteristics, and memory channel load, we can find the maximum $IPS/area$. Since the optimization space is discrete (other than the memory channel load) and relatively small, this can be done by exhaustive search. In the results below, on the order of 50 million different configurations are considered.

Table 4.2: Computational Complexity and Load and Store Frequencies of Workloads.

Workload W	$compl_W$	$f_{load,W}$	$f_{store,W}$
A - HPA	9.1	0.2319	0.0650
B - PPA	249	0.1691	0.0595

4.2 Workload and System Characteristics

In order to obtain results from the performance model, the parameters of the workload and the system characteristics have to be defined.

4.2.1 Network Processor Workload

For workload parameters in our model, the measurement results from CommBench in Chapter 3 are used, which include computational complexity, load and store instruction frequencies, instruction cache and data cache miss rate, and dirty bit probability. We aggregate the application parameters from CommBench into two workloads that we consider for the evaluation of our analysis:

- Workload A: Header-Processing Applications.
- Workload B: Payload-Processing Applications.

These workloads are such that there is an equal distribution of processing requirements over all applications within each workload. Table 4.2 shows the aggregate complexity and load and store frequencies of the workloads. The aggregate cache miss rates for instruction and data cache are shown in Figure 4.3. The x-axis corresponds to the effective cache size available to a thread as given in Equation 4.15. Note that we assume 2-way associative caches both for instruction and data cache. If different associativity was to be considered, the cache miss rates would have to be adjusted.

4.2.2 System Parameters

The system parameters for the network processor are listed in Table 4.3. The parameters that are varied in the optimization are chosen to cover a wide spectrum of configurations. The results below show that this covers the relevant regions of the design space. It is not expected that a wider range would result in any other optima. The values for the on-chip area of different components are approximate for .18μm

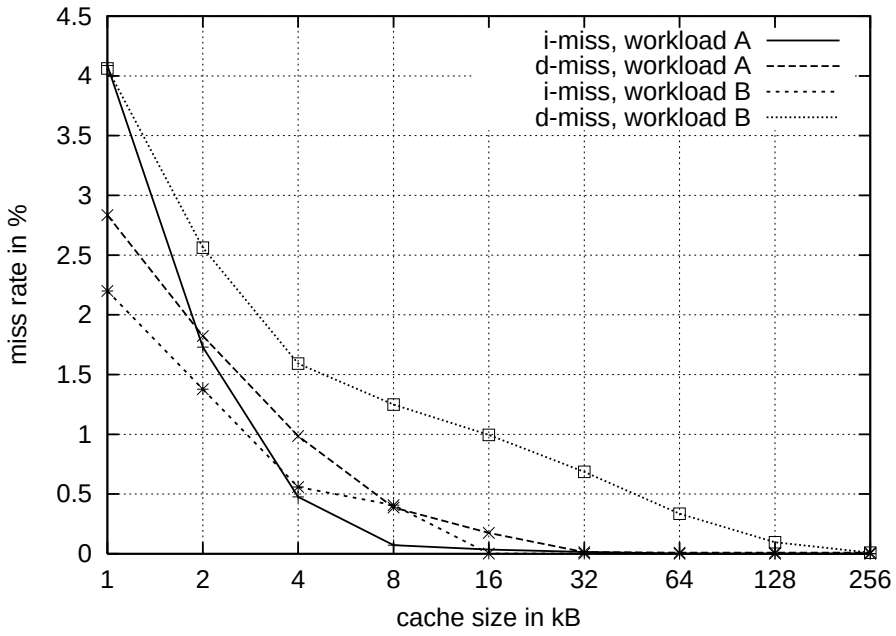


Figure 4.3: Aggregate Cache Performance of Workloads.

CMOS technology. It should be noted that exact values are hard to obtain from industrial sources. The performance model can of course be used with more accurate parameter sets. The total size of the NP is restricted to 400mm^2 or less, which is the limit of economic ASIC sizes.

4.3 Design Results

This section presents and discusses the optimization results and performance trends for the analytical model and parameter sets defined above.

4.3.1 Optimal Configuration

Table 4.4 shows the overall best configuration for both workloads. There are several important points that can be seen from this table:

- The optimal number of threads in both cases is $t = 2$, which indicates that it is not necessary to have a large number of threads to obtain good performance.
- The cache sizes are in the range of 16kB to 32kB, which yields an effective cache size of 8kB to 16kB per thread. These values correspond to knees in the i-miss

Table 4.3: System Parameters for Optimization.

Parameter	Value(s)
clk_p	200 MHz ... 800 MHz
t	1 ... 16
c_i	1 kB ... 1024 kB
c_d	1 kB ... 1024 kB
$linesize$	32 byte
τ_{DRAM}	60 ns
$width_{mchl}$	16 bit ... 64 bit
ρ_{mchl}	0 ... 1
$width_{io}$	up to 72 bit
ρ_{io}	0.75
$clk_{mchl},\ clk_{io}$	200 MHz
$s(p_{basis})$	1 mm^2
$s(p_{thread})$	0.25 mm^2
$s(c_i),\ s(c_d)$	0.10 mm^2 per kB
$s(mchl_{basis}),\ s(io_{basis})$	10 mm^2
$s(mchl_{pin}),\ s(io_{pin})$	0.25 mm^2
$s(ASIC)$	up to 400 mm^2

curves in Figure 4.3. Note that for the d-cache of workload B a small cache size gives better results since there is no clear knee in the curve that makes a larger cache pay off.

- Both configurations use the fastest processor because there is no cost in the model associated with higher clock rates. Also the widest memory channel is used, because it amortizes the basis cost $s(mchl_{basis})$ over a wider channel.

- The number of processors per cluster, n, is 31 and 20. This is relatively high, because a wider memory channel with more processors sharing it amortizes the basis cost better. When limiting the width of the memory channel to smaller sizes (e.g., 48 bit), the same configuration as in Table 4.4 with a smaller n (e.g., 24) and a larger m (e.g., 3) is the overall best. The $IPS/area$ value for this configuration is slightly lower (e.g., 173 MIPS/mm^2).

- The number of clusters per system is 2 or 3, which is limited by the overall chip area and the I/O channel width. With smaller memory channels and smaller n the number of possible cluster increases.

- The I/O channel is much wider for workload A because the processing complexity is much smaller for header-processing applications. Therefore data moves more quickly into and out of the network processor. For payload processing, the data remains on the processor for a longer time. Thus, a smaller I/O channel is sufficient.

- The overall processing power for both workloads is about the same (although workload B uses more chip area). Due to the lower complexity of header processing, this translates into a much larger throughput for workload A.

The most important result is the overall processing power, which is over 45000 MIPS. Considering the computational complexity of the workloads, this processing power translates to a maximum throughput of 42.48 Gbps for workload A and 1.48 Gbps for workload B. This shows that such configurations can easily handle gigabit link speeds even for complex payload processing.

4.3.2 Performance Trends

The optimal configurations of the network processor are very specific to a particular workload. To get more general results, we now look at the impact of different system parameters on the overall performance by varying them. Unless noted otherwise, parameters are fixed to: $t = 2$, $clk_p = 800$ MHz, $c_i = 16$ kB, $c_d = 16$ kB, $width_{mchl} = 64$ bit, and workload A. Note that these parameters correspond to the optimal configurations for workload A shown in Table 4.4. Also, ρ_{mchl} is chosen to be such that it yields the maximum performance. When using the term "performance," we mean $IPS/area$ (not IPS). Some of the configurations discussed below exceed the limits on total chip area, width of the I/O channel, and pin count. They are still shown as they might become feasible in the future.

Memory Channels

One critical parameter for the memory channel performance is the load, ρ_{mchl}. Figure 4.4 shows the performance of the network processor depending on the chosen load. It also shows the queue length given by the M/D/1 queuing model. For high loads the queuing time is so high that it impacts the performance of the processors. For most configurations the best load is about $\rho_{mchl} = 0.9$.

Table 4.4: Optimal System Configurations.

Parameter	Workload A	Workload B
clk_p	800 MHz	800 MHz
t	2	2
m	2	3
c_i	16 kB	32 kB
c_d	16 kB	16 kB
$width_{mchl}$	64 bit	64 bit
ρ_{mchl}	0.91	0.89
p_{miss}	0.187%	0.286%
τ_{mem}	137.6	121.6
ρ_p	0.974	0.957
n	31	20
$width_{io}$	71	3
$pins_{NP}$	199+$pins_{control}$	195+$pins_{control}$
IPS	48324 MIPS	45934 MIPS
$area$	272 mm^2	322 mm^2
$IPS/area$	178 MIPS/mm^2	142 MIPS/mm^2

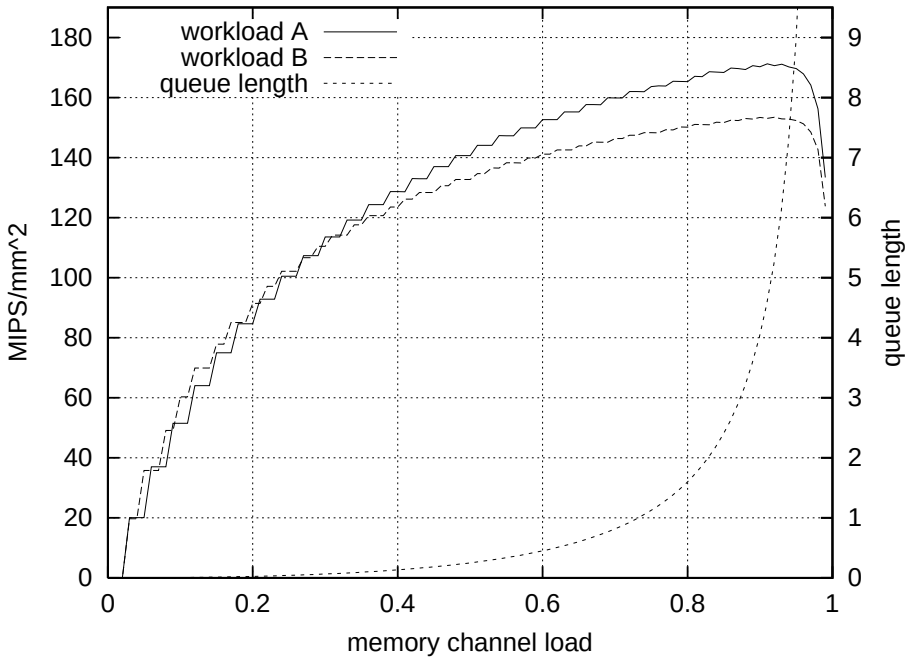


Figure 4.4: Performance Depending on Memory Channel Load.

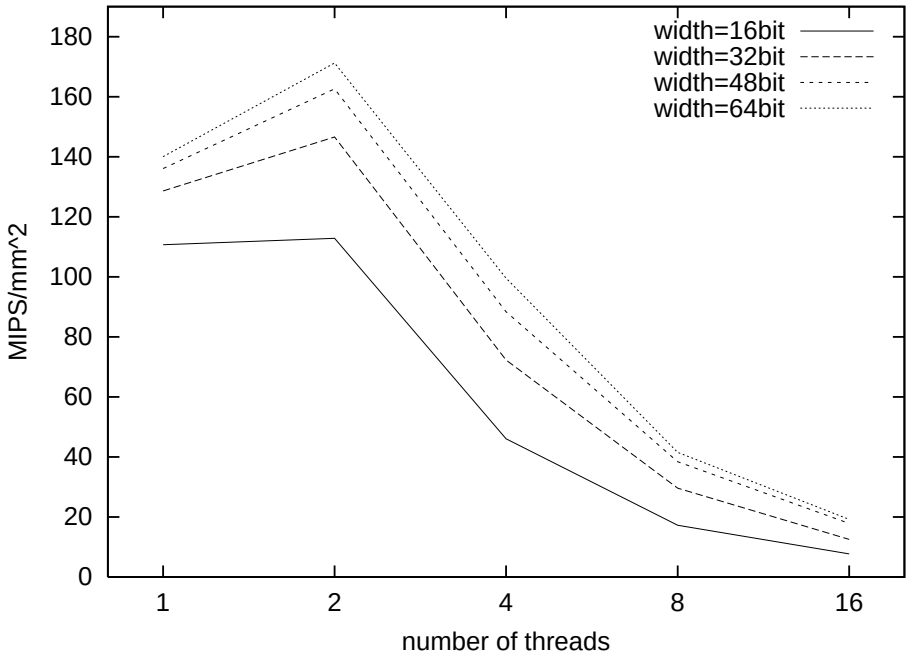


Figure 4.5: Performance Depending on Memory Channel Width and Number of Threads. The results for Workload A are shown.

The width of the memory channel also affects the performance of the network processor. Figure 4.5 shows that for one thread the memory channel performance does not impact the overall performance, because the system is mostly limited by τ_{DRAM} and τ_Q. For two or more threads, a four-fold increase in memory channel bandwidth (from 16 bit to 64 bit) yields up to twice the performance.

Processors

The processor can be configured in terms of clock rate and the number of thread contexts. Figure 4.6 shows the performance gains for higher clock rates over different numbers of threads. For one or two threads, the performance increases practically linear with clock speed. For larger numbers of threads, the amount of available cache per thread is less, which leads to more cache misses and possible memory stalls. Thus, the increase in performance is limited by off-chip memory accesses that cause processor stalls.

The performance impact of the number of available thread contexts can also been seen in Figures 4.5 and 4.6. In both graphs, the optimal number of threads is two. For larger number of threads, there are two factors that limit their benefits. One

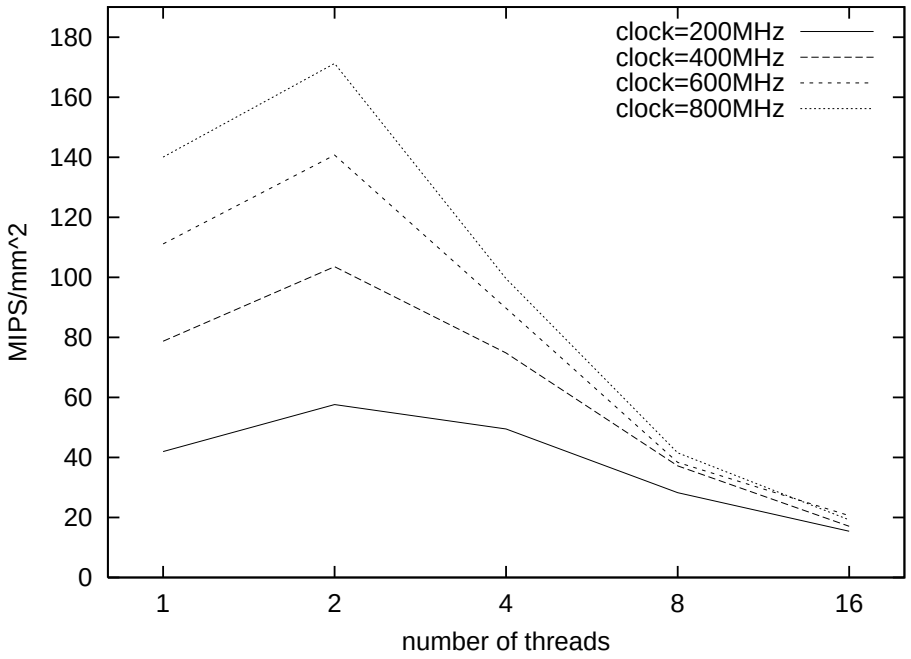


Figure 4.6: Performance Depending on Processor Clock Rate and Number of Threads. The results for Workload A are shown.

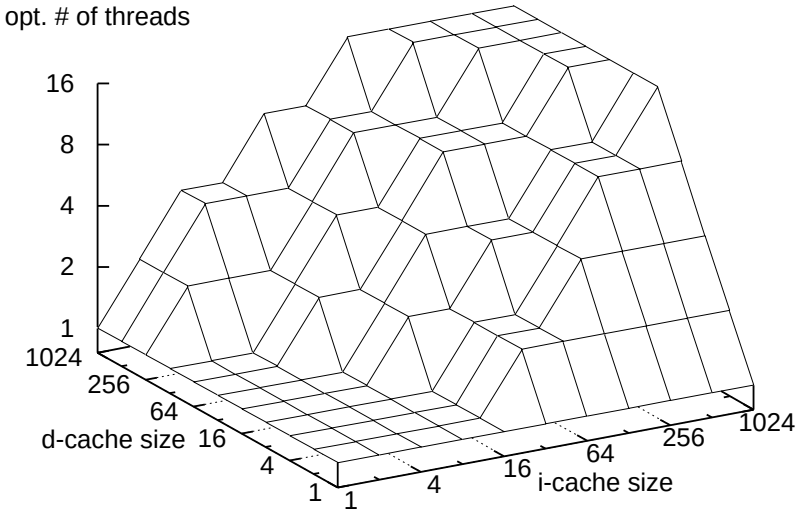


Figure 4.7: Optimal Number of Threads for Cache Configuration. The results for Workload A are shown.

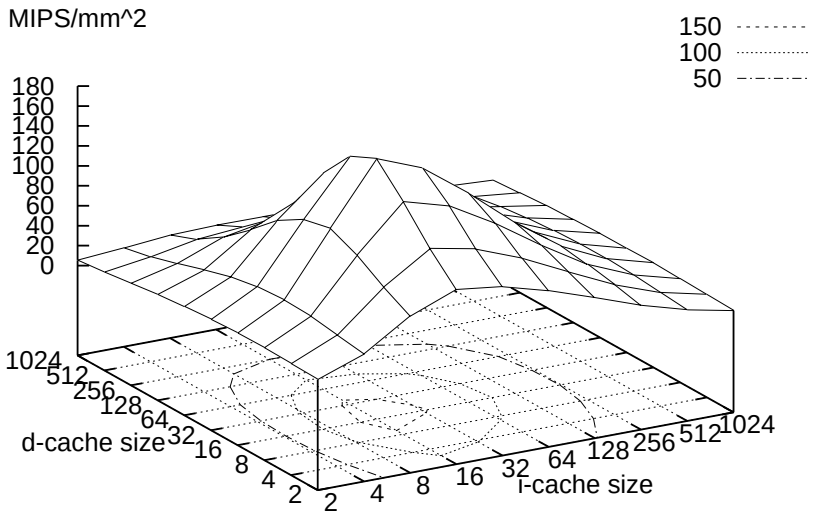


Figure 4.8: Performance Depending on Cache Configuration (Workload A).

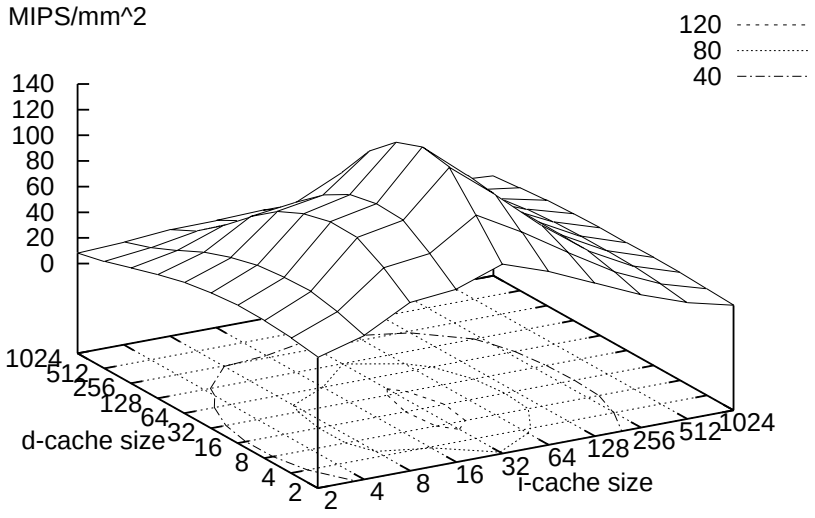


Figure 4.9: Performance Depending on Cache Configuration (Workload B).

is the higher cache miss rate due to memory pollution. The other is the additional area cost for the thread context.

To illustrate the impact of the cache pollution, Figure 4.7 shows the optimal number of threads for a given i-cache and d-cache configuration. This shows that if larger caches were available, more threads could be used for optimal performance. This indicates that with advances in on-chip memory technology, it can be expected that the number of threads in a processing engine will increase in the future.

Cache Memories

The size of on-chip caches is also an important configuration parameter. Since on-chip SRAM is expensive in terms of area cost, the amount of memory should be minimized, while still maintaining good cache hit rates to allow efficient execution of applications. Figures 4.8 and 4.9 show the performance of different cache configurations for both workloads. The performance is low for small caches due to high miss rates. It is also low for very large caches, since much chip area is used. The optimum for workload A lies at $c_i = 16$ kB and $c_d = 16$ kB. The optimum for workload B is at $c_i = 32$ kB and $c_d = 16$ kB. With $t = 2$, each thread uses effectively half of the available cache.

Another observation is that the performance is relatively sensitive to deviations from the optimal i-cache size. The d-cache size is less sensitive, but still has much impact on the overall performance. This leads to the conclusion that it is important to configure the memory system of network processors for the particular workload.

Chip Area Usage

Finally, to give a rough idea on how the chip area of a network processor is used, we evaluate what fraction of the total area is used for the processor (including thread contexts), the cache, and the memory and I/O channels. Figure 4.10 shows the fraction of processor area versus the fraction of cache area. The remaining fraction (to add up to 1) is the memory and I/O channel area. The top 1% (= 531) of all configurations are shown. Thus, the processor area typically makes up for 25-40% of the chip area. The cache area accounts for 20-60% and the memory and I/O channel area for 20-60%. The centroid lies at 34% for processors, 38% for cache, and 27% for memory and I/O channel.

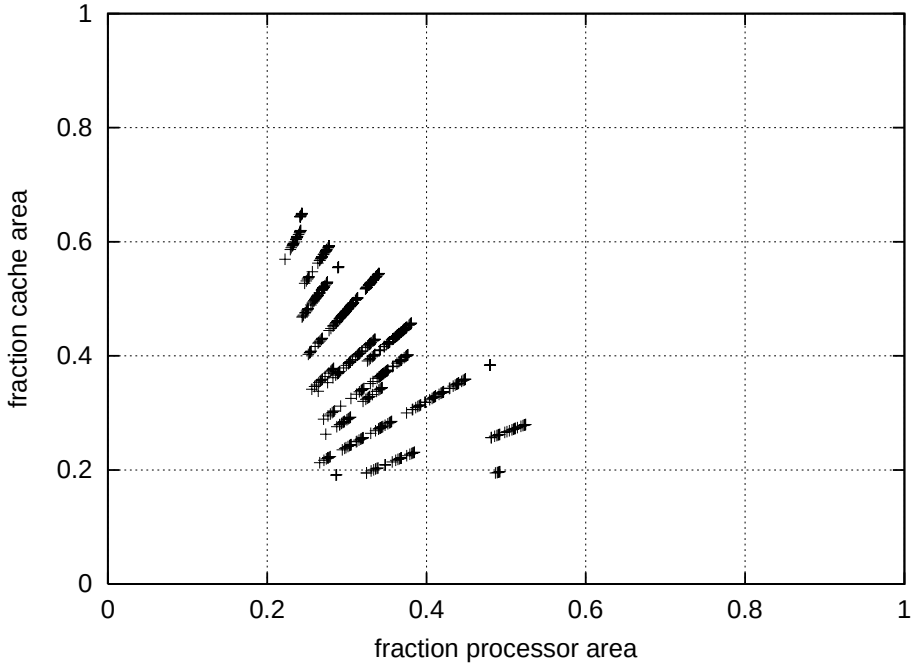


Figure 4.10: Chip Area Distribution for Top 1% of Configurations.

4.3.3 Sensitivity of Results

For the above results, a certain set of technology parameters were assumed as shown in Table 4.1. One important questions is how the results are affected when changing these parameters. This "sensitivity" of the optimum is shown in Table 4.5 for parameters that were not considered in Section 4.3.2. For each parameter, the optimal performance is shown for the default value as well as for a 10% lower and 10% higher value of the parameter.

Of the parameters considered, the optimal performance is most sensitive to changes in instruction and data cache area. For these parameters, a 10% change in cache area per kilobyte resulted in in a 3.79% change in the optimal performance per area. This is followed by the size of the processor core with about 2.3% impact for a 10% change. The thread size, memory and I/O channel size, and pin size have little impact with about 1%. The memory access time has no practical impact on the performance with only 0.16%.

These results indicate that the overall optimization results are relatively stable. Most sensitivity is exhibited when changing the cache area and processor core area parameters.

Table 4.5: Sensitivity of Optimization Results to Changes in System Parameters. Results are relative to the optimum of 177.7 MIPS/mm^2 for workload A from Table 4.4 (Opt. in MIPS/mm^2).

Parameter	Default		10% Decrease		
	Value	Opt. in MIPS/mm^2	Value	Opt. in MIPS/mm^2	Change in %
τ_{DRAM}	60 ns	177.7	54 ns	178.0	+0.16%
$s(p_{basis})$	1 mm^2	177.7	0.9 mm^2	181.8	+2.33%
$s(p_{thread})$	0.25 mm^2	177.7	0.225 mm^2	179.7	+1.15%
$s(c_i), s(c_d)$	0.1 mm^s/kB	177.7	0.09 mm^2/kB	184.4	+3.79%
$s(mchl_{basis})$, $s(io_{basis})$	10 mm^2	177.7	9 mm^2	179.7	+1.12%
$s(mchl_{pin})$, $s(io_{pin})$	0.25 mm^2	177.7	0.225 mm^2	181.0	+1.86%

Parameter	Default		10% Increase		
	Value	Opt. in MIPS/mm^2	Value	Opt. in MIPS/mm^2	Change in %
τ_{DRAM}	60 ns	177.7	66 ns	177.4	-0.16%
$s(p_{basis})$	1 mm^2	177.7	1.1 mm^2	173.7	-2.23%
$s(p_{thread})$	0.25 mm^2	177.7	0.275 mm^2	175.7	-1.13%
$s(c_i), s(c_d)$	0.1 mm^s/kB	177.7	0.11 mm^2/kB	171.4	-3.52%
$s(mchl_{basis})$, $s(io_{basis})$	10 mm^2	177.7	11 mm^2	175.8	-1.09%
$s(mchl_{pin})$, $s(io_{pin})$	0.25 mm^2	177.7	0.275 mm^s	174.5	-1.80%

4.3.4 Summary of Results

The above results of our performance model can be used to extract a few general design guidelines for network processors:

- The cache configuration has a big impact on the overall performance, which is sensitive to the workload.
- Two to four hardware contexts for threads is optimal. With large on-chip caches, more threads perform better.
- Higher processor clock rates and memory channel bandwidths are directly related to performance improvements for four or fewer threads.
- The chip area is split roughly evenly between processors, caches, and memory interfaces.

These results are somewhat dependent on the particular workload and systems parameter that are used. A more general rule of thumb is: "If a component in a network processor is not used efficiently, it might be better to use its area for another parallel processor." Nevertheless, the main contribution of this work is not the design results per se but the performance model that can be used with other workloads and system parameters.

4.3.5 Impact on Programmable Router Design

The performance model and the configuration optimization can be applied directly to the programmable router design discussed in Chapter 2. The APC configurations shown in Table 2.2 were derived using the above optimization. In addition, the performance tradeoffs shown above justify some of the design decisions of the APC:

- Multithreading. Figure 4.5 and 4.6 show that there is significant performance to be gained by supporting multiple threads. For both cases the optimum number of threads is 2. Figure 4.7, though, indicates that more contexts should be supported as larger on-chip memories become available in the future.
- Multiple Memory Channels. The access to off-chip memory is crucial as on-chip memory is limited in the APC design. To avoid heavy contention and long access delays (see Figure 4.4), it is important to have a design that is scalable in terms of the number of memory interfaces (i.e., clusters).

- Pin Count. The number of data pins for the APC as shown in Table 4.4 is only about 200. This shows that the total I/O for the APC (from packet transmissions and memory accesses) does not exceed typical configurations for ASICs.

Altogether, the results indicate that the overall router design is quite feasible for system-on-a-chip technology and can be used for implementing the processing infrastructure of a programmable router.

4.4 Related Work

Crowley *et al.* have evaluated different processor architectures for their performance under networking workloads [CFBB00]. This work mostly focuses on the tradeoffs between RISC, superscalar, and multithreaded architectures (as discussed in Section 2.3.1). In more recent work, a modelling framework is proposed that considers the data flow through the system [CB02].

Thiele *et al.* have proposed a very general performance model for network processors [TCGK02]. It takes into account the workload of the system in terms of data traffic streams, the performance of a processor under different scenarios, and effects of the queue system. The model is very general and could provide an interesting approach to network processor design. However, it still has to be shown that the results from this theoretical model can be applied to a realistic network processor system.

4.5 Summary

The network processor model and the associated performance expressions represent an attempt at developing a coherent approach to designing NPs. The model considers workload parameters, technology constraints, and a selection of design alternatives. The results can be used to optimize a particular configuration for a given workload. In addition, general performance tradeoffs can be derived to obtain a quantitative understanding of different design choices.

Chapter 5

Processor Scheduling Algorithms

General-purpose processing on a router introduces an additional level of complexity into the system, since not only link bandwidth, but also computational resources have to be allocated. While a significant amount of work has been done with respect to designing systems, which can provide guaranteed QoS to flows competing for bandwidth, processor sharing poses several new problems in this domain. Realizing such a system is fundamentally complicated by the fact that the execution times of various applications on packets are not known in advance, which limits applicability of well known bandwidth scheduling algorithms. Also, at a flow level, it is not clear as to how explicit or implicit admission control can be done as the processing requirements of a single flow are not known.

This chapter formalizes the scheduling problem and defines performance metrics for comparison of different scheduling algorithms. Measurements show that processing times can be estimated, which is helpful for resource reservation and scheduling. Two algorithms are presented: Locality-Aware Predictive Scheduling (LAP), which aims at reducing performance penalties due to cold caches, and Estimation-Based Fair Queuing (EFQ), which aims at enforcing fair sharing of processing resources. It is also discussed how both algorithms can be combined on the programmable router. This chapter is published in [WF01] and [PW02].

5.1 Scheduling Problem

Packets that are queued for processing need to be assigned to processors when these become available. The processor scheduler can choose any one packet from the n queues and assign it to any of the m processing engines if they are idle. This is

illustrated in Figure 5.1. After processing, packets are again queued in per-flow queues before the link scheduler in the QCTL assigns them to be transmitted on the link. The goal of scheduling is threefold:

- Good Performance. The scheduler should be work-conserving. That is if a processor is idle and a packet is available for the processor, the scheduler should not keep the processor idle. In addition, the scheduler should avoid "cold" instruction caches, which reduce the performance of the processing system. This effect is explained in more detail below.
- Fair Sharing of Resources. The scheduler should ensure that flows get access to processors evenly. That is the processor scheduler should ensure that no flow exceeds its fair share of processor usage.
- Low Delay. In order to minimize the effect of processing on the data flow, the scheduler should also aim at reducing the overall delay that a packet experiences. This also implies that the delay variation (i.e., jitter) should be minimized.

These goals are almost identical for scheduling in the link bandwidth domain, which has been studied intensively with many published solutions. However, there is one key reason, why these algorithms cannot be simply used for processor scheduling: In theory, processing time of an arbitrary piece of instruction code on a general-purpose processor cannot be determined beforehand (because it is a version of the Halting Problem for Turing machines). Most bandwidth scheduling algorithms rely on knowledge of packet sizes (which corresponds to transmission times on the link resource). Another reason is that transmissions of packets of the same size always take the same time. However in processor scheduling, the processing time of a packet depends on the packet data and the state of the processor as it was left by the previous packet (i.e., the state of the on-chip cache). Therefore it is necessary to consider new scheduling algorithms that take these issues into account.

5.2 Processing Characteristics

To illustrate the approach taken for developing the scheduling algorithms, this section discusses some characteristics of processing times on network processors.

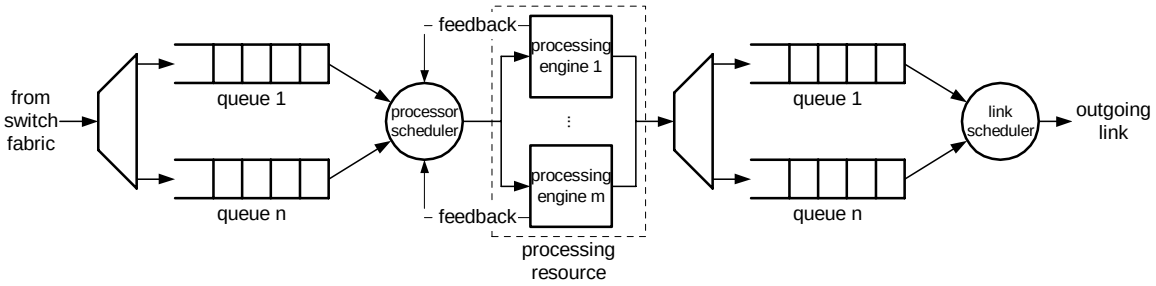


Figure 5.1: Scheduler System Outline.

5.2.1 Predictability of Processing Times

The nature of packet processing causes the applications to repeatedly execute the same code over the packets that are passed through the processor. This leads to good predictability of processing times as the following results show.

Measurements

Four applications were selected: encryption, compression, forward error correction, and IP forwarding. The first three applications are similar to payload processing applications presented in Chapter 3. For the measurements, the Washington University Gigabit Switch [CFFT97] enhanced with the single-processor linecard [DRST01] was used. The software environment for the processing utilized the Crossbow/Active Network Node operating system [DDPP98], [DPC+99] as discussed in Chapter 2. Several thousand packets were sent through the programmable router and the overall processing time for each packet measured. This process was repeated for different packet sizes and applications.

Figure 5.2 shows the processing time for packets of different sizes using the three applications. The error bars indicate the 95% percentile of processing time. For encryption and FEC, the processing times are very close to the average. For compression, which is a data dependent computation, the variations are slightly higher. Note that we use time as the metric for processing cost. This is done to simplify the description of the scheduling algorithm and its analysis. In a realistic network, processing cost should be translated to processor cycles per second and then adapted to the particular router system, where the packets get processed, as described in [GMC+00].

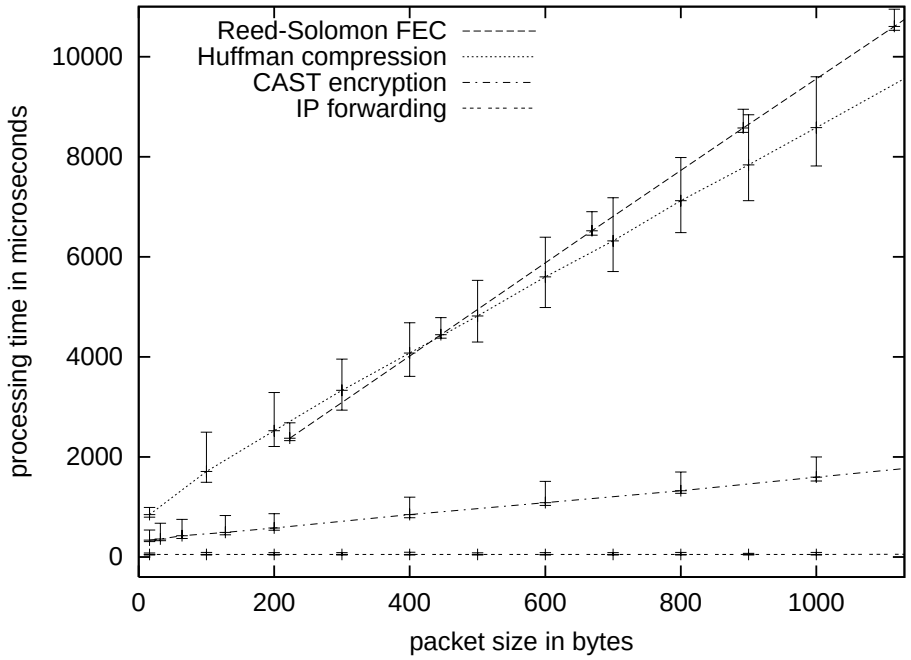


Figure 5.2: Packet Processing Times for Programmable Router Applications. The error bars indicate the 95%-percentile of processing times.

Processing Time Approximation

For IP forwarding, the processing time is practically constant for all packet sizes, which shows the per-packet processing cost of header processing. However, the processing times of the three payload processing applications are clearly dependent on the packet size. The per packet processing time for these applications can be extrapolated for packets of size 0. With these observations, we can define the estimated processing time t_e of a packet of length l when processed by application a as

$$t_e(a, l) = \alpha_a + \beta_a \cdot l, \tag{5.1}$$

where α_a is the per packet processing cost and β_a is the per byte processing cost of application a. Thus, the processing requirements of these applications can then be described by two parameters: α_a and β_a. These parameters for the four applications are shown in Table 5.1. To simplify notation, $t_e(p)$ be the estimated processing time of packet p, which is of length l and uses application a.

Table 5.1: Packet Processing Parameters.

Application a	per-packet cost α_a (μs per packet)	per-byte cost β_a (μs per byte)	cold cache penalty π_a (μs per pkt.)	expansion factor γ_a
IP forwarding	51	0	70	1
Encryption	320	1.3	170	1
Compression	970	7.6	950	0.13 - 0.34
FEC coding	320	9.2	175	1.14

Online Estimation

The parameters α_a and β_a in Table 5.1 have been determined from traces. But it is also possible to determine these parameters online and improve them using simple linear least squares regression techniques. As packets are processed the scheduler can maintain variables denoting the sums, $\sum t_{e,i}, \sum l_i, \sum t_{e,i}^2, \sum l_i^2, \sum(t_{e,i} \cdot l_i)$ for each application a. These variables are updated on the arrival of a new (c_{n+1}, l_{n+1}) pair and on completion of processing of a packet. The parameters to be used in the estimation can then be computed as

$$\beta_a = \frac{\sum_n t_{e,i} \cdot l_i - \sum_n t_{e,i} \cdot \sum_n l_i/n}{\sum_n l_i^2 - \sum_n l_i \cdot \sum_n l_i/n}, \tag{5.2}$$

$$\alpha_a = \sum_n t_{e,i} - \beta_a \cdot \sum_n l_i. \tag{5.3}$$

It should be noted that there are also applications, where the processing time cannot be as nicely correlated to packet size as shown above. An example for such an application is MPEG encoding. For MPEG encoding a whole video frame is required to perform effective compression. With unencoded video frames typically exceeding a packet size, processing can only be performed once several packets of a flow are buffered. In this case the processing time varies significantly between packets, but it can be expected to be more evenly distributed over frames (i.e., I-frame to I-frame). In such a case the parameters should be maintained for the group of packets constituting a single frame, which are always processed together.

5.2.2 Cold Cache Penalty

The optimization results from Chapter 4 show that on-chip cache sizes are typically small (16-32kB) due to die size limitations and can only hold data for the most recently

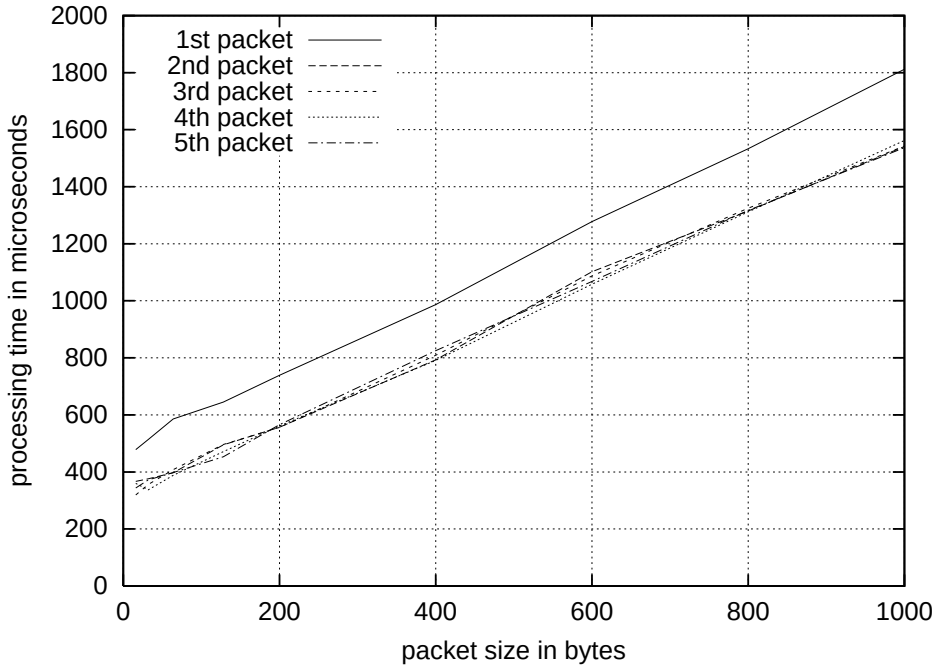


Figure 5.3: Cold Cache Effects on Packet Processing Times. The results are shown for the encryption application.

executed program. This data, which is mostly instruction code, can be reused by the processor if subsequent packets require the same program. Data cache information containing packet-dependent data can less easily be reused, since it changes with every packet. Changing the program that a network processor executes, causes the caches to become cold, which results in an execution time penalty associated with the initial loading of the cache with new application instructions. This can have a significant negative effect on overall network processor performance.

With the same measurement setup as above, this effect of "cold caches" can also be shown and quantified. These measurements are shown for the encryption application in Figure 5.3. When sending a stream of packets, which require the same application, the first packet encounters a cold cache. For subsequent packets, the processing time is reduced due to locality in the instruction code and the resulting warm cache. The measurements indicate that the cold cache penalty is independent of the packet size. Table 5.1 shows the average cold cache penalty, π_a, for all applications.

The Locality-Aware Predictive Scheduler discussed below aims at minimizing this cold cache penalty by assigning packets to processors that just completed processing the same application as required by the packet.

5.2.3 Reservations

A key component of quality of service is the definition of the service that is requested by a flow. While this is straightforward and well understood for link resources, reservations for computational resources are not as clearly defined.

Bandwidth Expansion

Processing of packets on routers can affect the size of the packets after the processing is completed. For many types of applications (e.g., encryption, routing lookup) the packet size is not changed, but a few applications can significantly change the bandwidth of a flow (e.g., compression, FEC). To take these changes into account, we define an expansion factor, γ_a, that is the average output bandwidth divided by the input bandwidth. This factor is also shown in Table 5.2. Note that the expansion factor can be dependent on packet size and data as for the compression application.

Admission Control

In an environment, where we want to be able to give service guarantees to data flows, it is typically necessary to explicitly reserve resources for that flow. This happens during the flow setup and allows the network to route a new flow in such a fashion that enough bandwidth is available on the chosen path. Now that we have shown that the processing requirements for a stream of data can be described in a simple manner, we can integrate this information into the flow setup process.

A reservation for a flow j with incoming bandwidth B_j that is processed by application a needs to reserve $\gamma_a \cdot B_j$ bandwidth on the outgoing link. The amount of processing P_j that is required (as fraction of one processor) depends on the bandwidth of the flow, the average size of packets l_j, and the application parameters:

$$P_j = \frac{B_j \cdot t_e}{l} = \frac{B_j}{l} \cdot (\alpha_a + \beta_a \cdot l). \tag{5.4}$$

Thus, flow j can be admitted to any router that has P_j processing power and $\gamma_a \cdot B_j$ outgoing bandwidth available.

Processing Location

When using reservations, it is necessary to determine the path of the flow and the location(s), where processing should happen. Ideally, the allocation of resources should

be optimal (e.g., best performance or lowest cost). Determining the best path in a traditional network can easily be done. However, with the additional processing step, it is necessary to develop a new approach. By combining the transmission and processing cost in a single metric and by modifying the network graph data structure to consider processing, we have shown that an optimal route can be computed efficiently [CTW01]. The details of this work are beyond the scope of this dissertation.

5.3 Locality-Aware Predictive Scheduling

The scheduler bases the decision of which packet to process next on control information that is received from the queue controller and the processors on the APC. The QCTL can provide information on the size of each enqueued packet and which application is required. The processors feed back information on when they become idle and which application was executed most recently. The definition of the scheduling problem is as follows:

> Given a sequence of packets $p_1 \dots p_n$, associated processing application requirements $\cup_{i=1\dots n} a(p_i)$, and a set of identical processors and their associated caches $u_1 \dots u_m$: Find a sequential assignment of processing units $u_i \leftrightarrow p_j$ $(i = 1, \dots, m; j = 1, \dots, n)$ to packets that maximizes a given performance metric (defined later).

The schedule, $S(u_t, Q_t)$, is a function of the set of packets in the queue memory, Q_t, at time t and the processing unit, u_t, which has become idle at time t. The assignment of a packet to a processor can be developed as a function of packet size, application properties, time, and state of the processors (see Table 5.2). Naturally, a schedule S is prohibited from assigning more than one packet to a processor u at any given time.

5.3.1 Scheduling Algorithm

The execution time of a packet depends on the state of the cache of the processor when it is processed. A cache is said to be cold if the application required by a newly assigned packet differs from the application just completed. If the cache is warm (i.e., not cold), the processing time is $t_a(p)$. If the cache is cold, a penalty of $\pi_a(p)$ is added to the processing time $t_a(p)$. The Locality-Aware Predictive (LAP) scheduling algorithm considers this. To compare LAP's performance, we also define

Table 5.2: System Parameters.

Component	Symbol	Description
packet p	P	the set of all packets ($p \in P$)
	n	number of packets ($\|P\| = n$)
	p_i	the i^{th} packet in the data stream
	$s(p)$	size of packet p
	$a(p)$	application a that is used to process packet p
application a	A	the set of all applications ($a \in A$)
	k	the number of all applications ($\|A\| = k$)
	$t_a(p)$	the actual processing time of packet p
	$t_e(p)$	the estimated processing time of packet p with warm caches
	$t_{cc}(p)$	the cold cache penalty for packet p
processing unit u	U	the set of all processing units ($u \in U$)
	m	number of processors ($\|U\| = m$)
	$W_t(u)$	set of apps for which processor u has a warm cache at time t
queue memory Q_t	Q_t	the set of all packets in the queue memory at time t ($Q_t \in P^b$)
	q	number of buffer slots ($\|Q_t\| = q$)
schedule S	$S(u, Q_t)$	the packet from Q_t that is assigned to u under schedule S
	$t_S(p)$	time when packet p is scheduled for a processor by schedule S
	$c(S(u, Q_t))$	returns 1 if assigned processor has cold cache, 0 otherwise
	$o_S(p)$	returns the order of packet p under schedule S

Throughput-Optimal (T-Opt), which is optimal in terms of least cold caches, and First-Come-First-Serve (FCFS), which is optimal in terms of least delay variation (as defined below).

Locality-Aware Predictive (LAP)

The locality-aware, predictive scheduling algorithm aims at making use of locality, while keeping the delay of the individual packets low. At each scheduling decision, LAP computes the fraction of processing that is necessary for each application based on the packets in queue memory. To achieve that, LAP uses an estimation of the processing time, $t_e(p)$, for each packet p. Define $f_{Q_t}(a)$ as the fraction of processing required by application a:

$$f_{Q_t}(a) = \frac{\sum_{\{p \in Q_t | a(p)=a\}} t_e(p)}{\sum_{p \in Q_t} t_e(p)}. \tag{5.5}$$

This fraction is compared to the fraction of processors that are currently executing application a (which means that they have a warm cache for application a). Let $w_t(a)$ be that fraction for a:

$$w_t(a) = \frac{|\{u \in U | a \in W_t(u)\}|}{m}. \tag{5.6}$$

Given $f_{Q_t}(a)$ and $w_t(a)$, LAP attempts to ensure that the fraction of processing power associated with applications (i.e., $w_t(a)$) is close to the that required by the packets in the buffer (i.e., $f_{Q_t}(a)$). LAP chooses to continue processing the application a for which u has a warm cache if changing the application would drop its processing fraction, $w_t(a)$, below the required fraction of processing, $f_{Q_t}(a)$. Thus, if $w_t(a) - \frac{1}{m} < f_{Q_t}(a)$, LAP picks the oldest packet with $a(p) \in W_t(u)$ from Q_t. Otherwise it picks the oldest packet overall.

$$S_{LAP}(u, Q_t) = \begin{cases} \arg\min_j \{p_j \in Q_t | a(p_j) \in W_t(u)\}, & if\ w_t(a) - \frac{1}{m} < f_{Q_t}(a) \\ \arg\min_j \{p_j \in Q_t\}, & else \end{cases} \tag{5.7}$$

LAP tries to group processors such that each group processes one application and thus keeps a warm cache for this application. The size of each group is determined by the amount of processing pending for packets in queue memory. The effectiveness of LAP is based on the assumption that the processing time for packets is predictable

from their size and the application they execute. LAP performance also depends on the number of packets available in queue memory and that the scheduler is aware of. We define this number as $|Q_t| = q$. Below, we can see that LAP performance increases with larger q as LAP can choose from a larger set of packets.

Throughput-Optimal (T-Opt)

We define Throughput-Optimal (T-Opt) as the algorithm that achieves maximum locality (and thus maximum throughput) by being allowed to pick any packet out of packet stream P (independent of Q_t). T-Opt executes all packets of one application before it switches the processor to another. Thus, the only cold caches are due to compulsory cache misses for the first packet of an application.

$$S_{T-Opt}(u, Q_t) = p_i, \quad where\ p_i = \arg\min_j \{p_j \in P | a(p_j) \in W_t(u)\}. \tag{5.8}$$

This strategy, though not realistic for actual implementation, gives an upper bound on the possible performance.

First-Come-First-Serve (FCFS)

A simple, basic scheduling scheme is first-come-first-serve (FCFS). In this scheme, packets are assigned to processors in the order of their arrival. If a processor u becomes available at time t, the oldest packet in queue memory Q_t is sent to u:

$$S_{FCFS}(u, Q_t) = p_i, \quad where\ i = \arg\min_j \{p_j \in Q_t\}. \tag{5.9}$$

The schedule does not take any locality into account. It is optimal in terms of variation in delay for packets since it does not re-order packets and keeps the delay for each packet in a given flow the same.

5.3.2 Evaluation

In order to evaluate the performance of LAP, several performance metrics need to be defined.

Performance Metrics

The performance of a schedule S can be defined in several (sometimes conflicting) ways. The performance depends in large part on the order of packet execution and the resulting processing time for the packet set. We define the following performance criteria:

- Throughput $T_S = \sum_{i=1\ldots n} s(p_i) / \left(t_S(p_n) - t_S(p_1)\right)$.
  The throughput is defined as the amount of data (i.e., $\sum s(p_i)$) that is processed in a given amount of time. This is the key performance parameter, since generally network processors are aimed at processing as much data as possible. Note that for simplicity, the execution time remaining after scheduling the last packet is ignored, since it has negligible effect on the results when n is large.

- Fraction of cold caches $C_S = \sum_{i=1\ldots n} c(S(u, Q_{t_i}))/n$,
  where $t_1 \ldots t_n$ is the sequence of times when scheduling decisions occur. The fraction of cold caches is the number of times a packet p is assigned to a processor with a cold cache (i.e., $c(S(u, Q_t)) = 1$) divided by the total number of scheduled packets. C_S is an indicator of how much locality awareness a scheduling scheme shows. The lower the fraction, the fewer cold cache penalties are incurred.

- Delay variation $D_S = \sqrt{\sum_{i=1\ldots n}(i - o_S(p_i))^2}$,
  where $o_S(p_i)$ is the order in which schedule S assigns packets to processors. If packet p_5 is the seventh packet to be processed, then $o(p_5) = 7$. Thus, for in-order processing $D_S = 0$. If packets are processed out of order, D_S is the standard deviation of the variation in the order. The larger D_S, the more variation, which means that certain packets are kept longer in queue memory, which increases their overall delay. While it is necessary to change the order of packet processing to make use of locality in reducing the negative cold cache performance effects, the goal is to keep D_S low. This will both reduce delay, and help to avoid large-scale re-ordering of the packet stream.

Using these performance measurements, the different scheduling strategies are evaluated below.

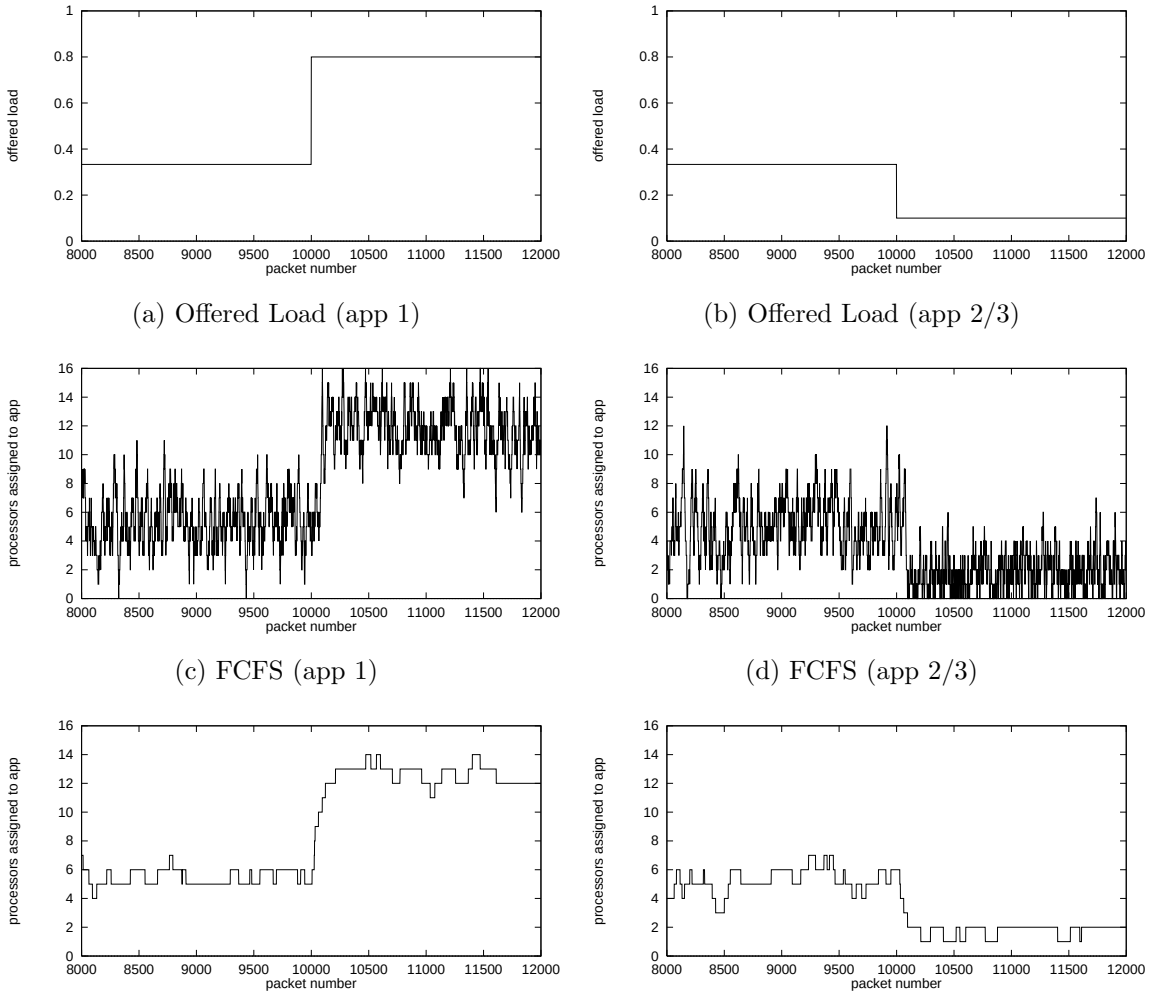
Simulation

The evaluation of the scheduling algorithms is done using a trace-driven simulation. Packet traces that are obtained from the packet processing time measurements described above. Traces of 100,000 packets are generated having an equal share of bandwidth for each application. To simulate more than three applications, the original traces are replicated with different application identifiers. We assume that a processor can only have one application in its instruction cache at any time, which is reasonable for the small cache sizes considered. These traces are used as input to a discrete event simulator that emulated the behavior of the scheduler and the processors. Simulations are performed over a variety of configurations. The number of processors ranges from 1 to 64, the number of packets in queue memory from 1 to 512 packets, and the number of applications in a packet trace from 3 to 300.

Basic Operation and Adaptation to Workload Changes

To illustrate the basic operation of each of the algorithms, we look at the case where we have three applications, 16 processors, and 64 packets in queue memory. The application workload is such that the first 10,000 packets require equal processing. Thus, each application on average should be processed on one third of the processors. After 10,000 packets, the workload changes, such that application 1 requires 80% of the processing and applications 2 and 3 require 10% each (see Figure 5.4(a) and 5.4(b)). This is used to illustrate the adaptability of the various algorithms to changes in the workload. Figures 5.4(c)-5.4(f) show the different scheduling algorithms. The lines show how many processors have warm caches for each application (i.e., how many processors process each application at that moment) for packets 8,000 through 12,000.

Each change in the number of assigned processors (y-axis) causes a cold cache, which reduces the overall performance. FCFS scheduling shows the expected "random" behavior. Since packets are scheduled in the order of arrival, no locality is explicitly exploited and the number of processors executing a given application changes frequently. This behavior leads to a large number of cold caches and low performance. A smooth scheduling behavior is produced by LAP scheduling, because it partitions the processors according to the processing requirements. Figure 5.4(e) and 5.4(f) show that the partitioning follows very closely to the offered load as shown in Figure 5.4(a) and 5.4(b).



(a) Offered Load (app 1)

(b) Offered Load (app 2/3)

(c) FCFS (app 1)

(d) FCFS (app 2/3)

(e) LAP (app 1)

(f) LAP(app 2/3)

Figure 5.4: Processor Assignment Comparison between FCFS and LAP. The scheduling assignments for applications 2 and 3 are similar and only one set is shown.

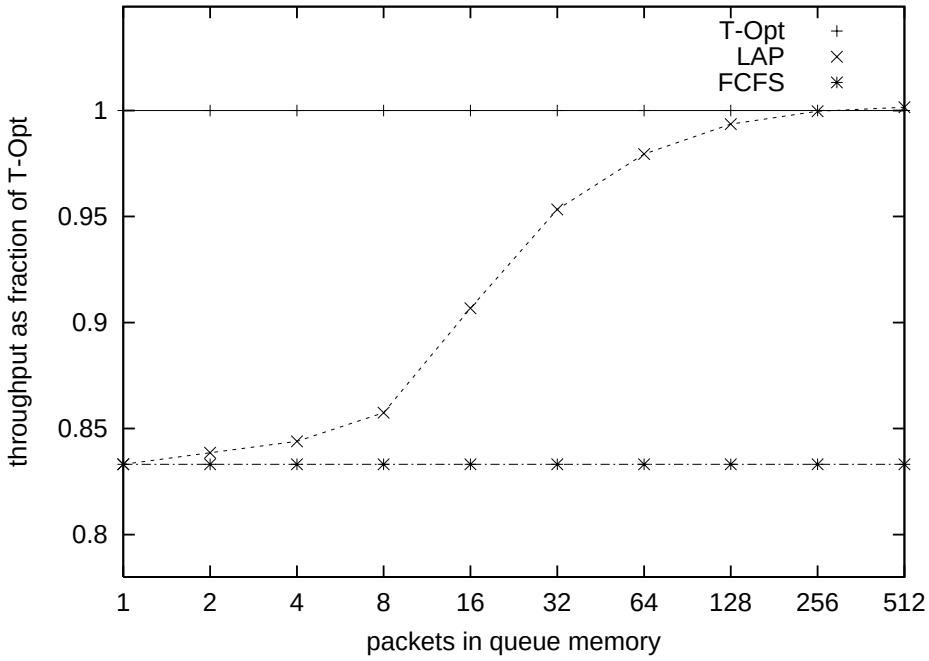


Figure 5.5: Throughput for LAP compared to FCFS and T-Opt. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

Also, LAP adapts quickly to changes in the workload. LAP reaches a processor assignment that corresponds to the offered load within a few hundred packets of the change in workload (3 to 4 times the number of packets in queue memory). During this period, packets from before the change are still in queue memory and influence the scheduling decision.

Throughput

Figure 5.5 shows a throughput comparison of LAP with FCFC and T-Opt. The number of processors considered is 16 and the number of applications is 30. Since LAP depends on the number of packets in queue memory, this value is varied on the x-axis. FCFS has the lowest throughput of about 85% of T-Opt. This can be expected, since FCFS does not take locality into account. For a very small number of available packets, LAP is close to FCFS, since the number of packets from which the algorithm can select is small and locality can only be maintained for short times. With about 16 to 64 packets, LAP performs significantly better than FCFS. For large numbers of packets, LAP converges towards the throughput of T-Opt.

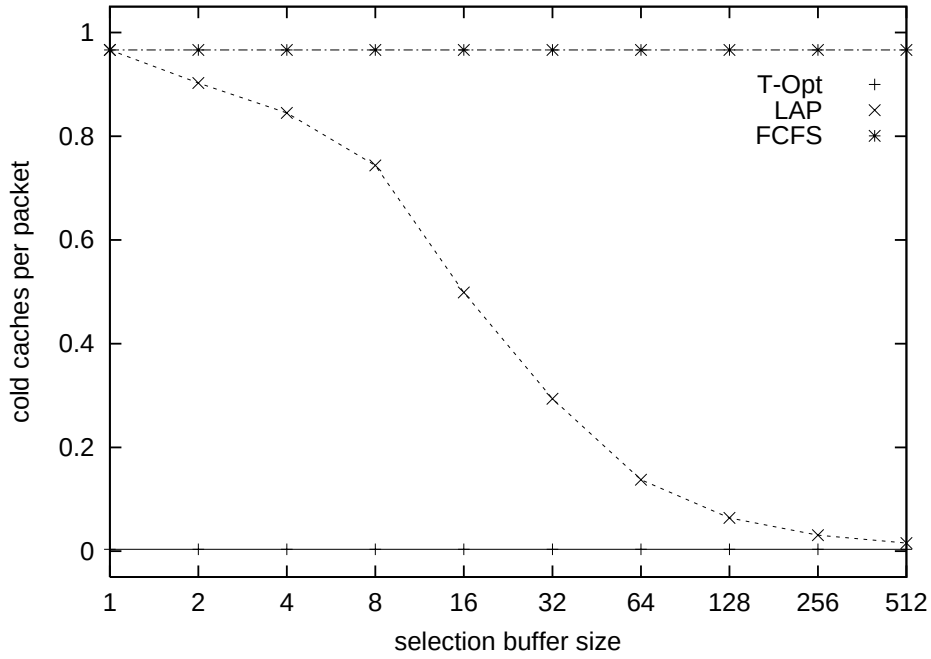


Figure 5.6: Cold Cache Fraction for LAP compared to FCFS and T-Opt. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

Cold Cache Fraction

To illustrate the correlation between the use of locality information and throughput, Figure 5.6 shows the cold cache fraction of packets for the same parameters as used in Figure 5.5. The cold cache fraction gives the percentage of packets that are executed with a cold cache (i.e., do not make use of locality). FCFS has the highest rate of cold caches with about 96%. This is due to the random assignment of packets to processors in FCFS, which causes only 1 in 30 assignments to be to a processor with warm caches (because $a = 30$).

The cold cache fraction for LAP shows a trend that corresponds to the throughput performance shown in Figure 5.5. For small numbers of available packets, the number of cold caches is close to that of FCFS. As more packets are available, cold caches drop and LAP converges towards T-Opt.

Delay Variation

Figure 5.7 shows the standard deviation of the variation in packet order for FCFS and LAP. The delay variation for T-Opt is arbitrarily large and thus not plotted

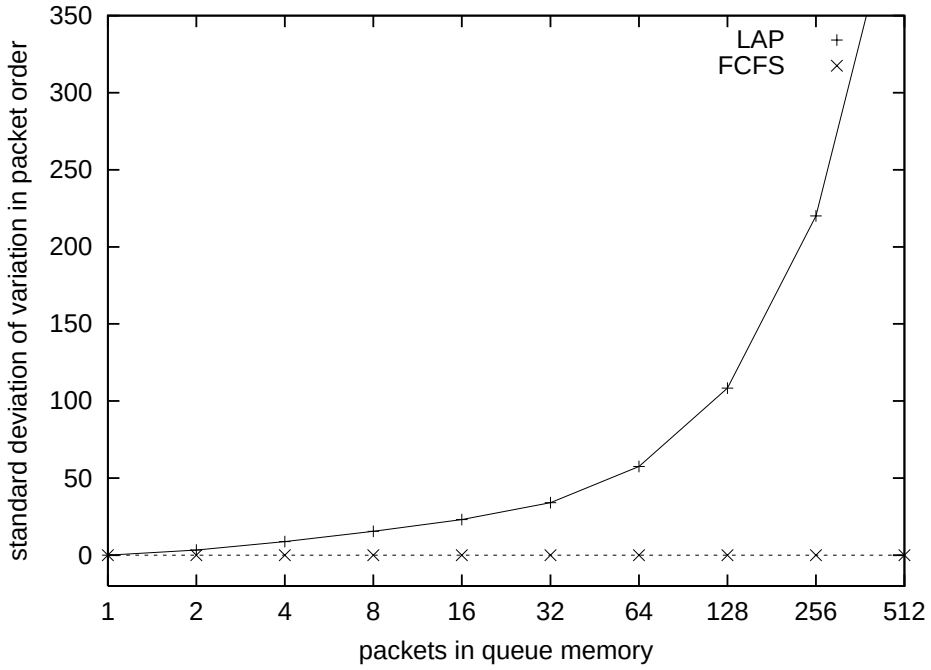


Figure 5.7: Delay Variation for LAP compared to FCFS. The number of available packets in queue memory is varied from 1 to 512 packets (30 applications, 16 processors).

here. For FCFS, there is no variation, because packets maintain their order. One can see that LAP shows increasing delay variation for increasing numbers of packets in queue memory. This is expected since the reordering is roughly limited to the number of available packets. The large variation of delay for increasing numbers of packets indicates that there is a tradeoff between achieving more cache locality and delay variation.

LAP Complexity

Finally, the usefulness of these scheduling algorithms depend on how efficiently they can be implemented in hardware. LAP has constant processing cost per packet, making it well suited for high performance systems. The following briefly discusses a possible data structure for LAP that can be implemented in hardware and has $O(1)$ update complexity.

There are three components necessary for LAP scheduling: the current values of $f_{Q_t}(a)$ and $w_t(a)$, a list of packets pending processing for each application in order of packet age, and a list of all packets in order of packet age. Each of these structures can be updated in constant time when a packet is received or scheduled.

The update of $f_{Q_t}(a)$ can be done every time a packet is entered into the buffer by adding its expected processing time. When a packet is removed, the processing time is subtracted. Similarly, $w_t(a)$ can be adjusted by incrementing and decrementing as processors change the applications that they process. An update occurs only when a packet enters or leaves the buffer. Thus, the complexity is $O(1)$ per packet. Maintaining lists of packets for different applications that are sorted by the age of the packets can also be done in constant time. Since the age of packets corresponds to the arrival order, a simple queue can be used. Updates to queues can be done in $O(1)$ time per update. There has been much work done in implementing efficient queuing systems of this sort [CT98].

Evaluation Summary

The evaluation indicates that good throughput performance can be achieved if 32 or more packets are available to chose from. However, the delay variation increases significantly causing large-scale re-ordering of packets in the data stream. Therefore, the throughput and delay seem to have a "sweet spot" around 16 to 32 available packets. To operate the system in this region, the number of packets to chose from could be limited artificially to 16 or 32. If more packets are available, the scheduler can only chose from the 16 or 32 oldest packets in the queue.

Another critical observation is that the algorithm causes the system to perform better as it becomes more loaded. For small numbers of packets (i.e., low load), the scheduler might cause a few cold caches. For more packets, the number of cold caches approximates the optimum.

In summary, LAP is a good scheduling algorithm for a system that is backlogged. LAP improves the throughput of the system by avoiding cold caches. Its only drawback is the delay that it might incur on packets. To address the issue of giving bounds on packet delay, we consider another scheduling algorithm.

5.4 Estimation-Based Fair Queuing

In contrast to LAP, Estimation-Based Fair Queuing (EFQ) aims to provide bounds on packet delays. This requires that EFQ operates in a regime, where admission control is performed, flows reserve bandwidth and processing, and the routers are operated below maximum load. EFQ is built upon the class of rate-proportional servers, which

have desirable properties that allow the use of processing time estimates to design a processor scheduling algorithm.

5.4.1 Scheduling Algorithm

The processor scheduler can view each processing engine as a separate resource to be scheduled if they individually have capacities exceeding the requirements of any single flow. The scheduler can also consider all the processing engines as a single processing resource, which can be scheduled using multi-server variants of single server scheduling algorithms [BO01]. In either case, the essential problem reduces to designing an efficient scheduling algorithm for sharing a single processing resource. This is what we consider.

We provide mechanisms for such a system to give guaranteed bandwidth and computational resources to incoming flows. Guarantees in these two dimensions mean that a flow always gets its reserved shares except when:

- A flow requires computational resources in excess of its reserved capacity and hence only a fraction of the incoming traffic is processed and forwarded to the link scheduler, possibly giving the flow a lesser share of its reserved bandwidth.
- Or equivalently, a flow exceeds its link share resulting in too many packets being queued up at the link scheduler, which forces the processor scheduler not to give the flow its processing share.

In order to do this, we base EFQ on the design methodology of Rate Proportional Servers.

Rate Proportional Servers

Definition Rate Proportional Servers (RPS) are a class of scheduling algorithms designed according to the methodology presented in [SV98], which allows the designer to trade fairness of the algorithm with implementation complexity. Generally speaking, a rate-proportional server is a work-conserving server with the following properties:

- The server has an associated system potential, which is updated to reflect the total work done by the server.

- Each flow in the system has an associated potential. When a flow becomes backlogged, its potential is set equal to the system potential. When a flow is already backlogged, its potential is updated to reflect the normalized service received from the server.

By imposing conditions for the potential functions as given in [SV98] and by serving packets from flows such that at any instant the individual potentials of all backlogged flows are equal, it can be shown that rate proportional servers have delay and fairness properties comparable to GPS. WF^{2}Q+ [BZ96] is an important example of a scheduler belonging to the RPS class.

We build on this methodology in designing the EFQ processor scheduling algorithm for two important reasons. First, the methodology helps in designing algorithms with delay bounds and fairness comparable to GPS without the complexity of GPS emulation. More importantly, the methodology provides us with enough flexibility to decouple the update of system potential from the exact finish times of the packets in the queues, which addresses the problem of not knowing the exact processing times in advance.

Packet Selection Policy A scheduling algorithm with optimal fairness would have to schedule single processing cycles according to the fluid Rate Proportional Server. However, in network processors, the smallest unit of processing is a complete packet. Context switching between packets is not considered here, because saving and recovering processing state is a relatively expensive operation compared to the short overall processing time for a packet. Thus, to approximate a fluid RPS, packets should be scheduled in order of their finish time with the earliest finish time first. While this works perfectly fine for bandwidth schedulers, the lack of the knowledge of the actual execution times of the packets, makes an exact implementation infeasible for processor schedulers.

However, to derive an approximate scheduler of this class, we can generalize the definition of a packet-by-packet RPS. Such a scheduler schedules two packets, j and k, of flows A and B, in the order in which they are more likely to finish processing. That is, if F_a^j and F_b^k are random variables representing the finish times of these packets in the fluid RPS, then packet j is scheduled for service before k, if

$$P(F_a^j \geq F_b^k) \geq 0.5. \tag{5.10}$$

Hence, it is the knowledge of the distributions of F_a^j and F_b^k which determines the accuracy with which schedulers can approximate GPS even if they use the same potential (or virtual time) functions. Also, since the potentials of individual flows are updated according to the normalized service received by the flows from the system, the finish time F_a^j is

$$F_a^j = P_a + \frac{W_a^j}{R_a}, \tag{5.11}$$

where P_a is the potential and R_a is the rate of service reserved by flow A. While these are known in advance when determining F_a^j, W_a^j, which represents the service time required by packet j, is not. Thus, the random variable F_a^j is directly determined by W_a^j.

Start-time Fair Queuing (SFQ) [GVC96b] (with a modified system virtual time) and WF^{2}Q+ [BZ96] are scheduling algorithms belonging to this class that represent the extremes with respect to the amount of knowledge of F_a^j. SFQ does not use any information about the service time of a packet and hence, according to the above policy, SFQ schedules packets in increasing order of P_a, which makes it suitable for processor scheduling. WF^{2}Q+, on the other hand, assumes that the exact service times of all packets are known in advance and thus determines the right order of servicing packets with probability 1.

Misordering Delay Different schedulers using the same potential functions and ordering packets for execution according to the above defined policy can give varying delays to flows based on their knowledge of the random variables W_a^j. To quantify these delays, assume that a scheduler of this class can be characterized by random variables χ_{a^j,b^k}, which denote the event that the scheduler (with its knowledge of W_a^j and W_b^k) makes a mistake in ordering packets j and k. That is, $P[\chi_{a^j,b^k} = 0]$ is the probability that the scheduler orders the packets of these two flows correctly, while $P[\chi_{a^j,b^k} = 1]$ is the probability that the scheduler makes a mistake in the ordering. Then, the average misordering delay, δ_a, as seen by a packet of flow A is the additional delay caused by the scheduler misordering packets of flow A and flow B, which is

$$\delta_a = P[\chi_{a^j,b^k} = 1] \cdot \frac{R_b}{R} \cdot (P_b + \frac{W_b^j}{R_b} - P_a - \frac{W_a^i}{R_a}). \tag{5.12}$$

This accounts for the time spent by the server in servicing additional traffic from flow B before processing packet from flow A. It is these additional delays caused by

misordering of packets that we intend to reduce using the estimates of the packet execution times we derived in Section 5.2.1, which improves the schedulers knowledge of W_a^j.

Estimation-Based Fair Queuing

Estimation-based Fair Queuing (EFQ) is a scheduling discipline designed for processor schedulers that uses the estimates of the packet execution times in ordering packets of various flows for processing. While the packet selection policy of any Rate Proportional Server can be changed to use these estimates, EFQ is derived by modifying WF^{2}Q+ which is known to have the tightest delay bounds and low time-complexity among bandwidth schedulers. EFQ, like WF^{2}Q+, uses a notion of system virtual time (system potential), defined by

$$V(t+\tau) = max(V(t)+\tau, min_{i \epsilon B(t+\tau)} S_i), \tag{5.13}$$

where $B(t)$ represents the set of backlogged flows at time t and S_i the start-tag associated with flow i as defined below. The above definition of $V(t)$ makes WF^{2}Q+ a Rate-Proportional Server. It differs from SFQ, in that it has a linear component, which ensures that the delay bounds provided are within one packet servicing time of a corresponding GPS server [BZ96].

For each flow i in the system, EFQ maintains a start tag, S_i (potential of flow i), a finish tag, F_i, and an estimated finish time tag, EF_i. Consider a packet k of flow i, with a reserved rate r_i, that arrives at time a_i^k. When this packet reaches the head of the queue, S_i is updated using

$$S_i = \max(F_i, V(a_i^k)), \tag{5.14}$$

if queue i is empty, else

$$S_i = F_i. \tag{5.15}$$

EF_i is updated using

$$EF_i = S_i + \frac{t_e(p_i^k)}{r_i}, \tag{5.16}$$

where $t_e(p_i^k)$ is the estimated number of instructions required to process packet k (see Equation 5.1). When the processor finishes processing this packet, the actual finish

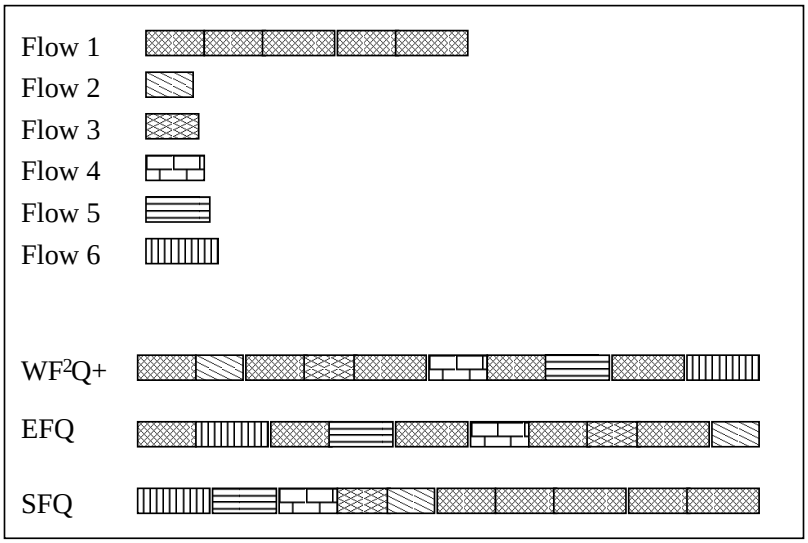


Figure 5.8: EFQ Scheduling Example. All flows have backlogged packets of the same length and are processed by the same application. The figure shows the actual execution times of packets as their size and the processing order derived by different scheduling disciplines.

tag F_i is updated using feedback from the processor:

$$F_i = S_i + \frac{A_i^k}{r_i}, \tag{5.17}$$

where A_i^k is the actual number of instructions required to process packet k. This ensures that each flow is correctly charged for processing time, even if the initial estimate was incorrect.

Given these tags, the EFQ scheduler, schedules packets in increasing order of their estimated finish time tags EF_i.

Example

The following illustrates the behavior of EFQ and compares it to that of SFQ and WF^{2}Q+. Consider a set of flows, all of which send packets of the same length but at different rates and are processed by the same application. Figure 5.8 shows six such flows, with flow 1 reserving 50% of the processing resource and the rest of the flows reserving 10% each. The size of a packet in Figure 5.8 represents the *actual* processing time of that packet. Note, however, that the *estimates* for all packets are the equal, since they all have the same length and are processed by the same application.

WF^{2}Q+ achieves an optimally fair schedule, because it is assumed the scheduler knows the actual processing times. Thus, the packets of flow 1 and the other flows alternate (due to the rate reservations). Out of flows 2-6, the packet of flow 2 is processed first, because it has the lowest actual execution time and therefore the lowest finish time.

EFQ expects all packets to have the same execution times. Thus, EFQ could pick any order of packets 2-6 to alternate with packets from flow 1. The worst case, which introduces most misordering delay, is shown in Figure 5.8. Here, the packet of flow 2 is processed after packets of flows 6, 5, 4, and 3 are processed, which all use more processing time than expected by scheduler. As a result, the packet from flow 2 experiences an additional delay due to the variation in actual processing times of these packets. However, these variations are much smaller (and bounded, for the applications in consideration) than the total processing times of the packets themselves. In particular, these delays are much smaller than those introduced by SFQ.

As shown in the example, in the worst case SFQ could delay the processing of the first packet of flow 1 until packets from all other flows are processed. This is due to all initial packets having the same start time.

In summary, EFQ processes most packets in the same order as WF^{2}Q+. When either a flow reserves a much higher rate than others or has greatly differing processing requirements (due to differing packet sizes or applications), the variations in the actual executions times compared to estimated execution times do not change the scheduling order. Even in the case when the scheduling order of packets in EFQ varies from that of WF^{2}Q+, the additional delay that is experienced by a packet is bounded by the variation in execution times as opposed to the total execution times of packets as in SFQ.

Analysis

From the example given above, it can be seen that for N flows, in the worst case, SFQ introduces a misordering delay of

$$\delta_{SFQ} = \sum_{i=1}^{N} \frac{A_i^{max}}{R} - \frac{A_a^{max}}{R_a}. \tag{5.18}$$

This is obtained by using $\forall k : \chi_{a^j,b^k} = 1$ with the misordered packets being of maximum size and using $\forall b : P_b = P_a$ in Equation 5.12, since the scheduler can make

a mistake only when $P_b \leq P_a$. Results below also show that SFQ actually favors (i.e., gives lesser delays to) flows with packets which require greater average normalized service (i.e., higher $\frac{t_e^{avg}(p_a)}{R_a}$).

To analyze EFQ, assume that for a given packet length, the packet execution time estimates obtained in Section 5.2.1 can be represented by uniform random variables W_a^j lying in the range $[t_e(p_a^j) - V_a^j, t_e(p_a^j) + V_a^j]$. The EFQ scheduler misorders packet j and k when it determines that

$$P_a + \frac{t_e(p_a^j)}{R_a} \geq P_b + \frac{t_e(p_b^k)}{R_b}, \tag{5.19}$$

but the actual processing times are such that

$$P_a + \frac{A_a^j}{R_a} \leq P_b + \frac{A_b^k}{R_b}. \tag{5.20}$$

In the worst case, we get

$$\frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} \leq P_b - P_a \leq \frac{A_a^j}{R_a} - \frac{A_b^k}{R_b} + \frac{V_a^{max}}{R_a} + \frac{V_b^{max}}{R_b}. \tag{5.21}$$

Hence from Equation 5.12, the misordering delay for packet j due to packet k is limited to

$$\delta_a = P[\chi_{a^j,b^k} = 1] \cdot \frac{R_b}{R} \cdot \left(\frac{V_b^{max}}{R_b} + \frac{V_a^{max}}{R_a} \right) \tag{5.22}$$

and the worst case misordering delay is bounded by

$$\delta_{EFQ} = \sum_{i=1}^{N-1} \frac{V_i^{max}}{R} - \frac{V_a^{max}}{R} + \frac{V_a^{max}}{R_a}. \tag{5.23}$$

From the above equation we can see that as the number of flows increases, δ_{EFQ} only increases with the variations in execution times as opposed to δ_{SFQ} which increases with total processing times. Also note that, with a better estimation, e.g., by including higher order moments in characterizing W_a^j, EFQ can more accurately determine the right scheduling order, resulting in a smaller δ_{EFQ} and thus approximating WF^{2}Q+.

5.4.2 Evaluation

In this section, we present simulation experiments to demonstrate the improved performance of EFQ as compared to SFQ.

Simulation Setup

To compare the delay characteristics of the two schedulers, we use the following simulation setup. First, we use the traces of actual execution times of packets from different flows that are processed by different applications on the programmable router. These traces are then used by a packet generator to feed the two simulated schedulers: SFQ and EFQ. The speed of the processor in the simulator is 2GHz (about 10 times the speed of the processor on the Smart Port Card (SPC) [DRST01] on which the actual measurements were made). The system has 32 flows with different packet sizes, which are processed by the four different applications. All the flows reserve the same processing rate and adjust their sending rates to just saturate their share of the processing resource. These flows together require just below 100% of the system's processing resources. Thus, they can all be admitted and the measured delays are only due to scheduling and not due to queuing backlog.

Packet Delay

Figure 5.9 shows the delays of various packets of a flow, which is processed by the forwarding application. The interarrival time of the packets of the flow is approximately 163 microseconds, which is just enough to saturate the flow's share of processing resources. Note the high and bursty delays experienced by the packets of the flow when scheduled by SFQ as shown in Figure 5.9(a). Since SFQ, always schedules packets with the minimum virtual time, a single packet of a flow can be delayed in the worst case by the equivalent of the sum of one packet processing time of all other flows. In the simulation this translates to a worst case misordering delay of 8218 microseconds. The maximum delay actually observed in Figure 5.9(a) is about 6100 microseconds, implying an observed maximum misordering delay of $6100 - 163 = 5937$ microseconds.

For EFQ, much lower delays can be seen in Figure 5.9(b). This illustrates two things. First, given the small execution time of forwarding as compared to other applications, the finish times of the packets of this flow where so different compared to the finish times of the packets of other flows that the errors in estimates did not change the scheduling order (i.e., Equation 5.21 was not satisfied for most comparisons of

finish times). Second, the worst case delay that could be experienced by these packets is only 1312 microseconds which would occur if there were maximum variations in the estimated execution times for packets from all other flows at the same time. In the simulation, the maximum misordering delay observed is about $900 - 163 = 737$ microseconds, which is about one order of magnitude smaller than for SFQ.

Figure 5.10 shows the delays experienced by a flow being processed by the CAST encryption application, with the average packet size of the flow being 200 bytes and has a higher average processing time per packet compared to the forwarded flow. While the average delays experienced by the packets when scheduled using EFQ is close to the interarrival time of the packets indicating a very low misordering delay, the average delays seen in Figure 5.10(a) are about three times the interarrival time of the packets. Figure 5.11 shows the delays experienced by a flow being processed by the FEC application which requires much greater processing time per packet compared to the above flows. Here, the average delays seen by the packets when scheduled by SFQ are actually less than the interarrival time of the packets. This indicates an average negative misordering delay, while delays due to EFQ are just about the interarrival time of the packets. Two important conclusions can be drawn from these plots:

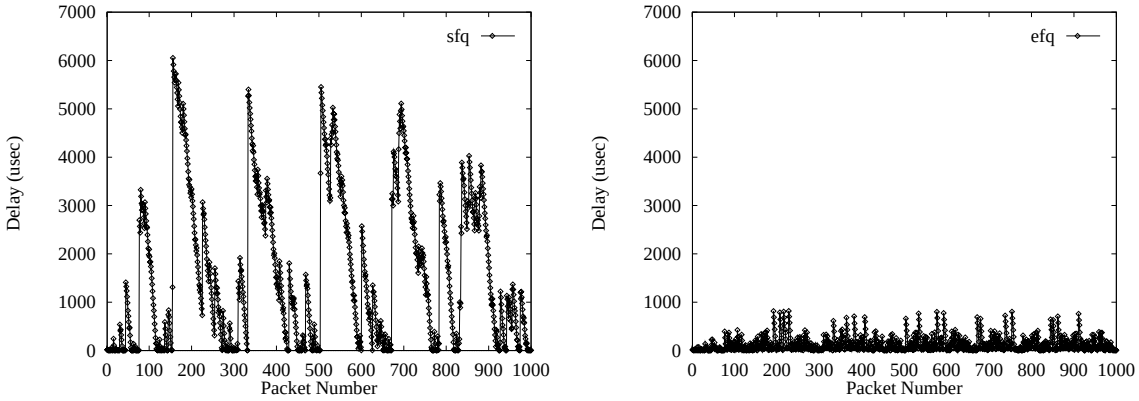
- SFQ gives much higher misordering delay bounds than EFQ.
- Across flows, while the misordering delays due to EFQ are on an average close to zero, they vary from high positive misordering delays (e.g., the delay of about 35 times the interarrival rate seen by the forwarding flow) to low negative misordering delays when scheduled using SFQ.

The second point indicates a bias of SFQ.

Biased Delay Bounds Due To SFQ

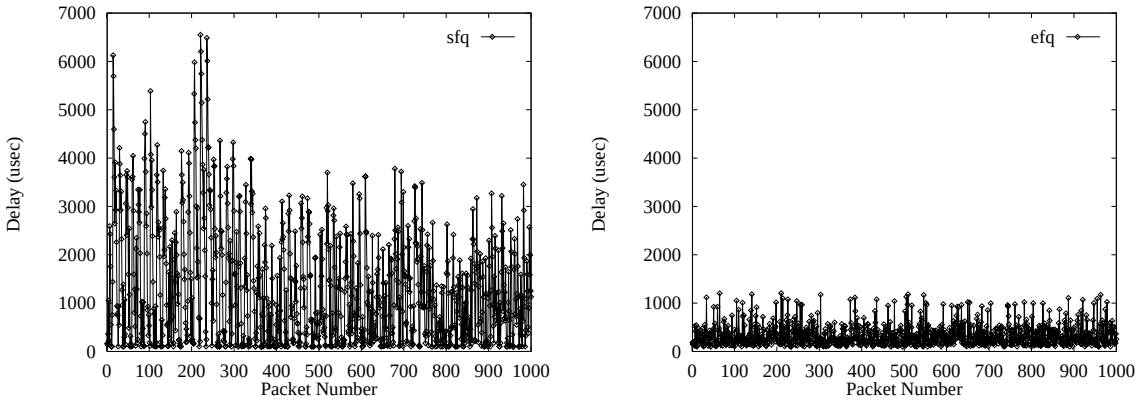
The bias of SFQ can be explained by the work conserving nature of the two schedulers. If SFQ gives high positive misordering delays to some flows, there should be flows in the system which get low and in fact negative misordering delays, while EFQ gives low (close to zero) average misordering delays for all flows. We actually show a correlation between the misordering delay experienced by the packets of a flow and the average processing time per packet to reserved processing rate ratio (i.e., $\frac{t_e^{avg}(p_a)}{R_a}$).

SFQ favors and gives less misordering delays to flows with higher average processing time to reserved rate ratio over flows with a lower ratio. Given a set



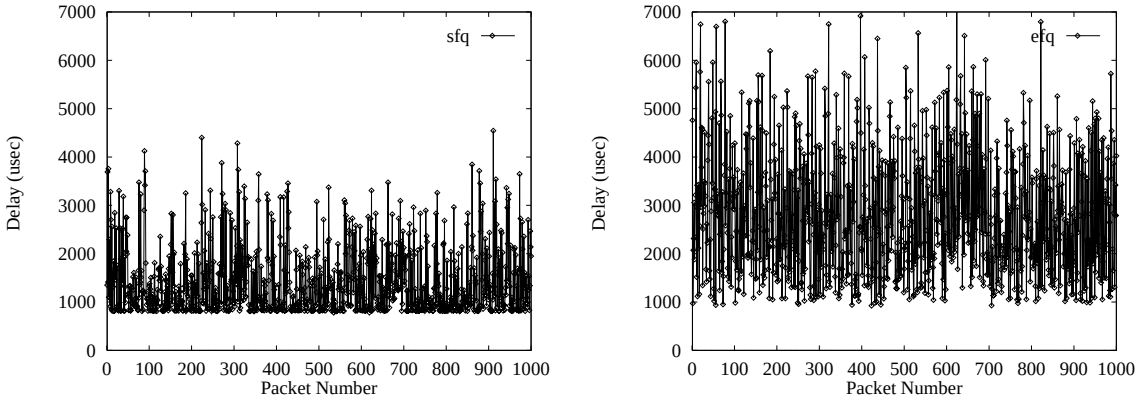
(a) SFQ (b) EFQ

Figure 5.9: Packet Delays for a Flow Processed by IP Forwarding.



(a) SFQ (b) EFQ

Figure 5.10: Packet Delays for a Flow Processed by CAST Encryption.



(a) SFQ (b) EFQ

Figure 5.11: Packet Delays for a Flow Processed by Reed-Solomon FEC.

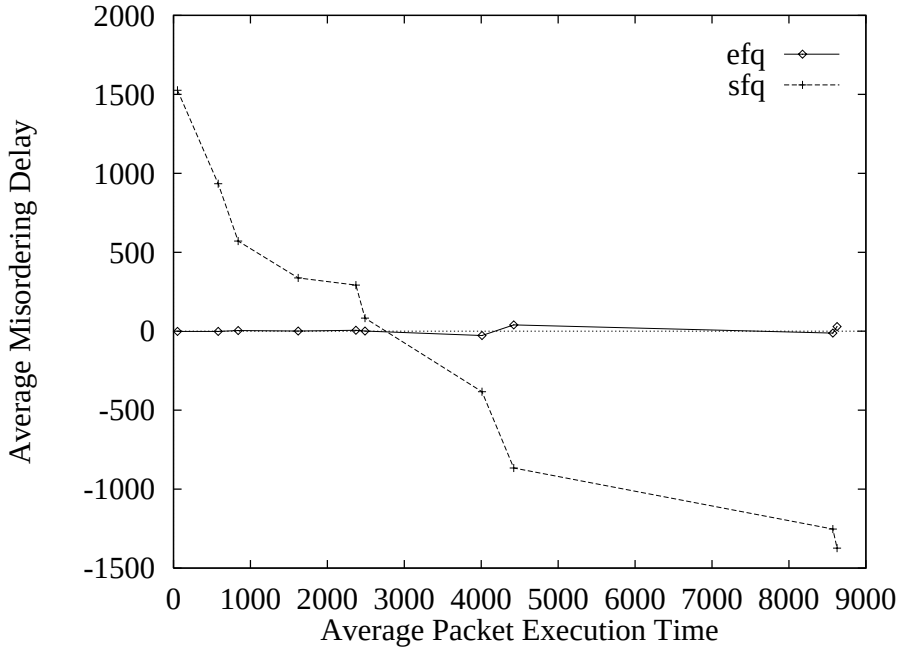


Figure 5.12: Variation in Minimum Packet Delay for Different Flows Introduced by SFQ and EFQ.

of flows with the same potential, since SFQ can schedule them in any random order, it is very likely that a packet of a flow with higher average processing time to reserved rate ratio is scheduled before at least a few flows with lower ratios, resulting in lower delays for such flows. EFQ by just using the estimates is able to rightly reverse this order. Figure 5.12 shows the average misordering delay introduced by the two schedulers plotted with increasing average packet execution times. Note that all the flows have the same reserved processing rates. This plot clearly shows the above conjectured correlation between average misordering delay and average processing time per packet to reserved rate ratio.

Simulation Summary

In summary, the simulation shows three main results. One is that the analytically derived worst case misordering delay is almost reached by the SFQ scheduler as shown in Figure 5.9(a). Second, EFQ shows a much lower and smoother scheduling delay. This is due to the delay depending on the variance of the processing times rather than the absolute processing times as in SFQ. Third, SFQ introduces unfairness by favoring flows with high processing time to reserved rate ratios. This behavior is not shown by EFQ, which provides fairness over a wide range of processing requirements.

5.5 Combination of LAP and EFQ

The LAP and EFQ scheduling algorithms address two different problems in network processor task scheduling. LAP increases the throughput of the system as it makes use of instruction locality. EFQ ensures fair sharing of resources and bounds the misordering delay of packets. Both schedulers also operate in different usage regimes of a programmable router. LAP performs best when packets are backlogged in queue memory, which is characteristic for a system, where the processing resource is a bottleneck. EFQ ensures fair sharing and delay bounds when the processing resource is not oversubscribed. It is also necessary that flows adhere to the specified data rate to avoid additional packet delay due to backlog.

These differences indicate that LAP should be used in an active network, where processing is performed on a best-effort basis and no explicit flow setup is performed. EFQ on the other hand requires explicit reservations, which are only available in a network that is tightly managed and where resources are controlled. Such a network could provide QoS in terms of bandwidth and processing resources.

It might be possible to combine the ideas of EFQ and LAP in a single scheduling algorithm. In particular, in EFQ, we assume that the processing resource is not oversubscribed. Therefore it is quite possible that at any point of time when a scheduling decision is necessary more than one processor is available to handle the scheduled packet. In such a case, locality information could be used to chose a processor, which has a warm instruction cache. This would reduce the processing time of a packet and allow more flows to be admitted to the router.

One drawback of such a scheme is that this could violate the QoS guarantees of EFQ. If the distribution of application usage changes over time, it might not be possible to assign processors with warm caches anymore. As a result the processing time of a flow increases and the processing resource might be overloaded. But in EFQ this should never happen. Other possibilities of combining EFQ and LAP are considered as future work.

5.6 Related Work

Cache-affinity scheduling, which uses locality information for the scheduling decision has been used mostly in shared memory multiprocessors [VZ91], [DM92], [SL93], [TTG95]. The focus in this domain is to schedule the same process or thread on

processors that can reuse previously established cache state. While this is similar to the network processor environment, it does not consider the reuse of instruction cache state for different threads that use the same instruction code (as is done with packets that use the same application).

An example for scheduling that uses hints about the processing requirement is [PEA+96]. In this work, the compiler provides information about thread requirements that are used by the scheduler to determine a thread execution schedule with high cache locality.

Salehi *etal.* show the effect of affinity-based scheduling on network processing in [SKT96]. While this also considers the processing of network traffic, the focus is on the operating system level, where packet processing is disrupted by a background workload. This switching between packet processing and the background workload reduces locality in execution and can be avoided by appropriate scheduling.

Most software-based programmable routers (see Section 2.5.1) enforce isolation of packet processing between flows (e.g., malicious packets cannot effect the proper processing of other packets). However, QoS issues at the level of processing are addressed only in a few cases. The commonly used NodeOS specification [Pet01] asks for packets to be processed by individual threads to allow for an accounting mechanism. However, methods for admission control and QoS scheduling are not described. Qie *et al.* [QBPK01] describe the problem of scheduling computational resources among competing flows, but relies on being able to pre-determine the processing time of packets. Also, the important issue of correlating the cycle rate of a flow to the bit rate is not addressed. There are also approaches where the expressiveness of the processing environment is restricted (e.g., no loops) to give execution time guarantees [MHN01], which limits its usefulness to simple header processing applications.

Packet service disciplines and their associated performance issues have been widely studied in the context of bandwidth scheduling in packet-switched networks [Zha95]. The performance of these disciplines has been compared to Generalized Processor Sharing (GPS) [PG92], which has been considered an ideal scheduling discipline based on its end-to-end delay bounds and fairness properties. Packet Fair Queuing (PFQ) disciplines, however, cannot be used for processor scheduling. PFQ disciplines like WFQ, WF^2Q [BZ95] use a notion of virtual time, whose correct update in a processor scheduler, requires precise knowledge of execution times of various packets in advance. Efforts have been made to design service disciplines which isolate

the scheduler properties that give rise to ideal fairness and delay behavior, without emulating GPS [Gol94]. Notable among these are a class of schedulers called Rate Proportional Servers [SV98], which decouple the update of system virtual time from the finish times of packets in queues. But even these service disciplines, while avoiding the complexity of GPS emulation, schedule packets in order of pre-determined finish times, which in turn requires the knowledge of execution times of various packets in advance.

An exception to these disciplines is Start-Time Fair Queuing (SFQ) [GVC96b], which has been deemed suitable for CPU scheduling [GVC96a]. Since SFQ does not need prior knowledge of the execution times of packets (packet lengths in a bandwidth scheduler), it is also applicable to scheduling computational resources. However, the worst case delay under SFQ increases with the number of flows and can in fact worsen in the presence of correlated cross-traffic as shown in [BZ96].

Our work is aimed at providing a way of estimating execution times of packets, which is used on a flow level for admission control and for QoS scheduling at a packet level.

5.7 Summary

This Chapter presents two algorithm for processor scheduling on a programmable router. We show that network processing applications exhibit very regular and predictable processing patterns, which helps overcome the obstacle of theoretically undeterminable computation times of arbitrary programs. The processing time estimations can be approximated by a linear function that we use for admission control. Locality-Aware Predictive (LAP) scheduling schedules packets such that instruction cache state on processors can be reused, which effectively reduces the packet processing time and increases the system throughput. The Estimation-based Fair Queuing (EFQ) algorithm uses the processing time estimates to fairly and efficiently assign packets to processing engines.

Chapter 6

System Simulation

A simulation of the proposed programmable router system was implemented as part of this work. The simulation is accurate on a cycle level and captures the behavior of the Application Processing Chip in detail. The Queue Controller, Flow Classifier, and other components are modelled at a behavioral level.

In this chapter, the data and control path of the simulator are presented in detail. Several memory management issues for packet storage and packet processing are discussed. Simulation results are presented and contrasted to analytic results, which have been derived in Chapter 4.

6.1 Introduction

The proposed programmable router introduces a set of features on the router port, which allow custom processing. Several of these require careful consideration as they are crucial for the overall system performance. The simulation of the router port addresses these issues and proposes a possible implementation. The simulation also allows the verification of analytic results.

Simulations can be modelled at different levels of detail. While it is desirable to be as detailed as possible, there is a cost in terms of implementation complexity and run-time performance. For this work, the level of detail is chosen to be cycle-level accuracy in the Application Processing Chip and behavioral accuracy with consideration of timing in other components. The following list highlights the features of the simulation:

- Actual Packet Transmissions and Processing. The simulation processes packet traces in *tcpdump* format [MJ93]. The packets of these traces are moved through the router port, processed by the APC, and stored on the output.

- Cycle-Accurate Processor Simulation. The processor cores on the APC are simulated using the *SimpleScalar* processor simulator [BA97]. This simulator is extended to capture the multi-processor nature of the APC.

- APC Programmability with High-Level Language. The programs for packet processing are compiled from C code. Using a "plugin template," new processing functions can easily be implemented

- Realistic Memory Management. The memory on the APC is simulated to keep state between packets of a single flow, share instruction code among flows using the same application, and keep packet data from different packets apart. This is important in order to obtain realistic cache performance results. Also, the queue memory is simulated with realistic data structures for packet storage.

- Timing for all Components. All parts of the router port are simulated considering processing, memory access, and communication delays. In particular memory interfaces consider contention, bandwidth limitations, and DRAM access times.

- Configurability. The simulator can be configured to consider a broad range of APC configurations and other system parameters.

There is also a set of issues that were simplified for this simulation. A more accurate implementation of the following is beyond the scope of this work:

- Flow Classification and Scheduling Decision Cost. While there is a fixed delay associated with flow classification and scheduling, it is independent of the number of flows and packets that are present in the system.

- Delays Synchronized with Clock. While the delays of various components are considered, they are not necessarily synchronized with a system clock. This should yield only minor differences in results.

- Access Patterns on DRAM Memory. The simulation assumes that all DRAM accesses can be interleaved in a split-transaction fashion and that there is no delay due to accesses to the same memory bank by different requests.

- Cache Coherence. The simulation does not allow for interaction between programs *during* execution. Once a program completes the packet processing its state changes are visible, but during execution it operates in isolation. As a result cache coherence is not an issue and therefore not considered. In realistic systems, however, this can have a significant impact on performance.

- Cache Write-Back / Write-Through. Contention on the memory channel due to write-backs or write-throughs from cache is not considered.

While these issues need to be considered in a real implementation of a programmable router, they are not expected to significantly impact the results of the simulation.

6.2 System Simulation

The simulation is implemented in an event-based fashion, where different components interact with each other over well-defined interfaces. The clear separation of components ensures a realistic movement of data in the system and consideration of all delays. The following describes in detail the data and control paths as well the processor simulation, queueing system, and other components.

6.2.1 Data Path

The overall data path of the simulation is shown in Figure 6.1. The Queue Controller and the APC are indicated by boxes around their respective components. Packet buffers and queues are indicated by shaded areas.

Packets are generated by the link interface and switch interface from stored traces. The flow classifier determines in which queue a packet should be stored and what type of processing it requires. Packets are then stored in queue memory, which consists of SRAM for meta-data and DRAM for the actual packet. More details on the queue memory are discussed below. Controlled by the processor scheduler, the APC interface forwards packets through the I/O channel to the APC. Processing is then done on the processor, where the execution trace is played back to generate an accurate pattern of cache accesses. After processing, the packet returns to the queue memory. The link scheduler then determines when it should be sent out to the link or to the switch. If the destination in the switch fabric is the same port as the current port, the packet immediately appears on the input side in the switch interface. Not

shown in the figure is the case where a packet does not require processing, in which case it is stored in queue memory and then sent to the output.

6.2.2 Control Path

The control path of the simulation illustrates the sequence of events that occur as a packet traverses the system. Figure 6.2 shows the graph of events. Each event is placed in the component, where it triggers an action. Not shown are the events for the input from and output to the switch, which are similar to those for the link. The various interactions are discussed in more detail in the following subsections.

6.2.3 Processing Engine Simulation

The APC is the main focus of the simulation. As discussed above it is meant to be cycle-accurate and perform the actual processing of packets. For this purpose, SimpleScalar, an established RISC processor simulator, is used to perform the processing.

Processors

The APC is a multiprocessor system, which requires multiple processors to be active at the same time. SimpleScalar per se does not support multiprocessors. Therefore, the APC simulation processes packets one-by-one, captures each instruction trace, and "plays back" the trace to generate accurate timing and caching behavior. For this purpose, the `sim-safe` instance of SimpleScalar was modified to generate a trace of instructions that is executed. For each instruction, the program counter, opcode, and memory access address is recorded. This trace is then used by the APC simulator.

For each active thread in each processor in each cluster, an instruction trace is stored. When the thread executes an instruction, the next entry from the trace is taken. Instruction fetch and possible load/store operations are then passed on to the cache to see if they can be satisfied. If a cache miss occurs, the thread is stalled and another thread takes over. Once the requested cache line is available, the original thread is "unstalled" and can continue processing when the processor becomes available. This behavior is illustrated in Figure 6.2, where the case of a cache miss is shown. The event PR_INTR_EXEC corresponds to the instruction execution. CA_ACCESS checks if a datum is available in cache. If not, PR_STALL_THREAD occurs, which stalls the current thread and switches the context to another available thread. Once

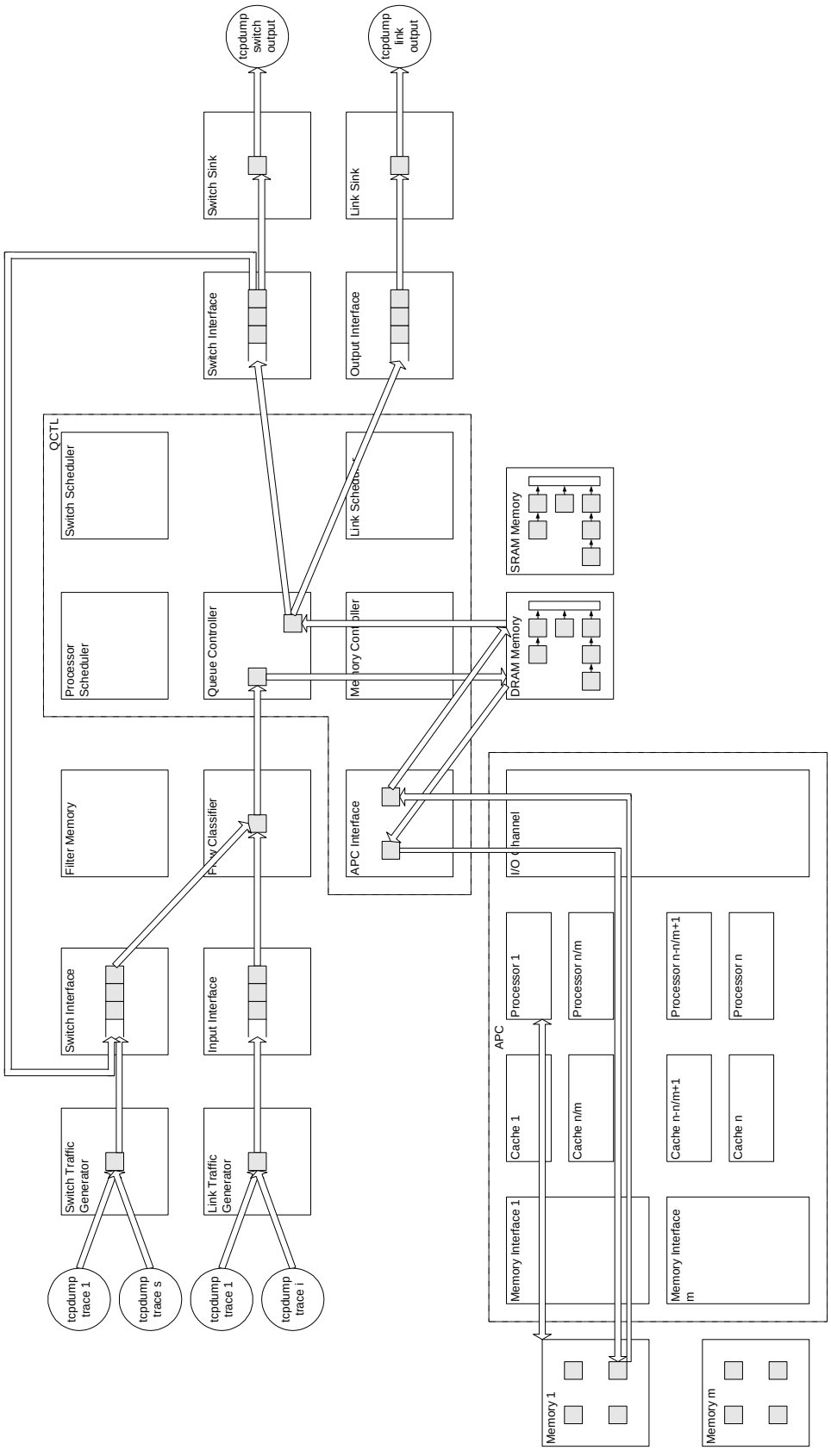


Figure 6.1: Simulation Data Path. Buffers and queues for packet data are shown as shaded areas.

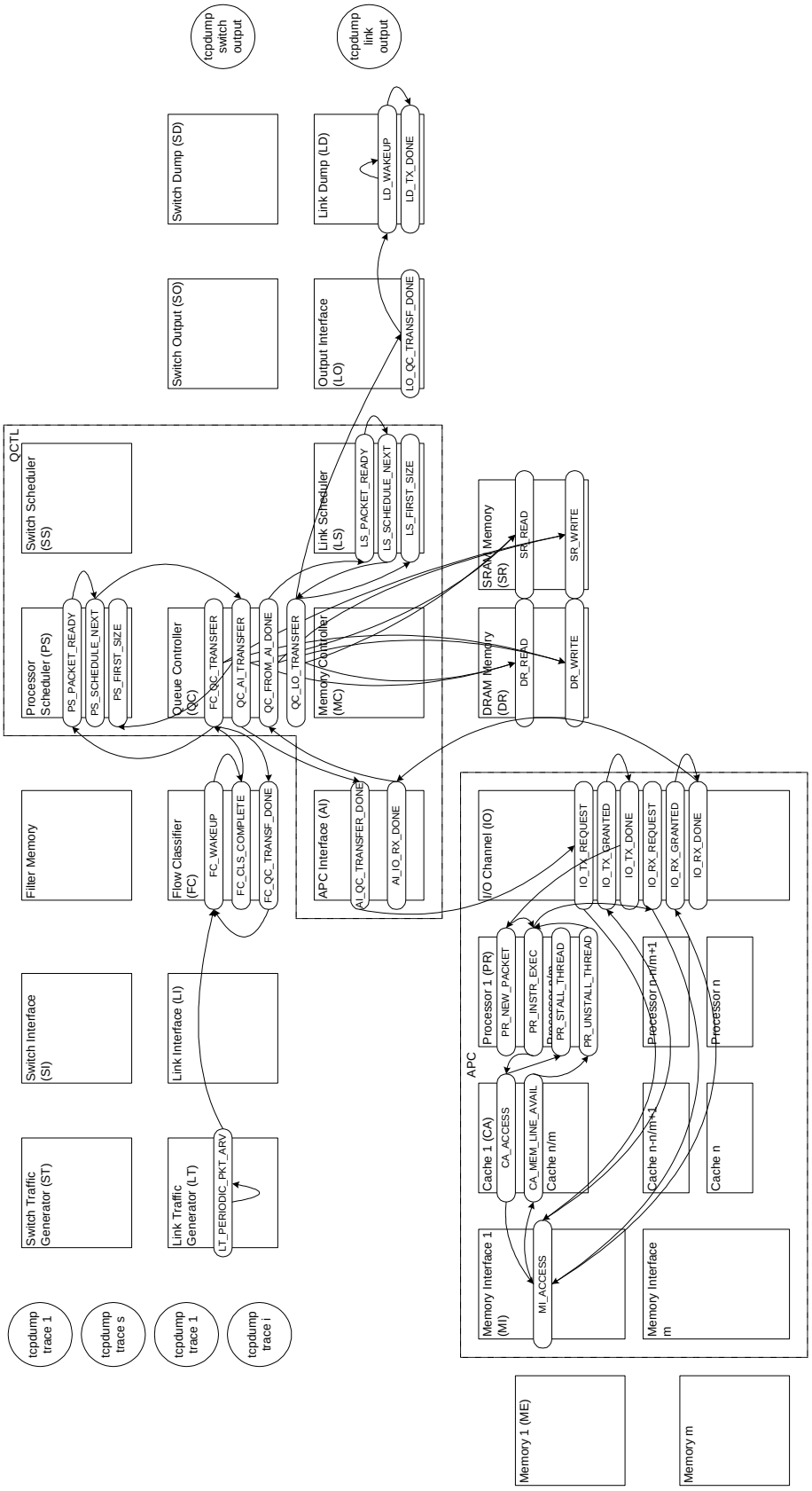


Figure 6.2: Simulation Control Path. Events that trigger actions in components are shown as ovals.

MI_ACCESS is completed, the cache line is available (CA_MEM_LINE_AVAIL) and the PR_UNSTALL_THREAD event allows the thread to continue processing.

While maintaining a complete instruction trace for a program can be expensive in terms of memory requirements, the processing of packets is typically limited to a few thousand instructions, which can be handled easily.

Cache Simulation

The cache simulation needs to determine if a memory access leads to a cache hit or a cache miss. For this purpose, a simple caching data structure is used. When an address is requested, it is checked if the appropriate cache line is available. Note that it is not necessary to maintain actual data in the cache, as this is already done in SimpleScalar. It is sufficient to keep the addresses of currently active cache lines to make a hit/miss decision. For the simulation, n-way associative caches (with configurable n) with a least-recently-used replacement policy are implemented.

The address space is shared among threads in processors, which allows the sharing of data. By partitioning the address space appropriately, the following behavior can be achieved for different types of data:

- Instruction Data. Program code should be shared among threads, which allows i-cache reuse.
- Flow Data. If threads process packets from the same flow, that data should be shared among them. Note that due to the way the simulation is implemented, there is no need to consider any possible hazards due to memory accesses by more than one thread.
- Packet Data. Packet data should only be accessible to the thread that currently is processing the packet. Also, when the packet changes, the thread should not be able to access data from the previous packet (e.g., in cache).

This partitioning is implemented such that each data address has an implicit identifier for the respective application, flow, and packet. This is illustrated in Figure 6.3. For example, the address space for instructions of application 5 is from 0x0005000 to 0x0005ffff. This allows for 4096 different applications with 64kB instruction data each. For flow data, 1024 flows with 256kB of flow-state each are supported. This amount of per-flow state should be enough to support even more complex applications, like video transcoding. This address space also includes the execution stack, which grows

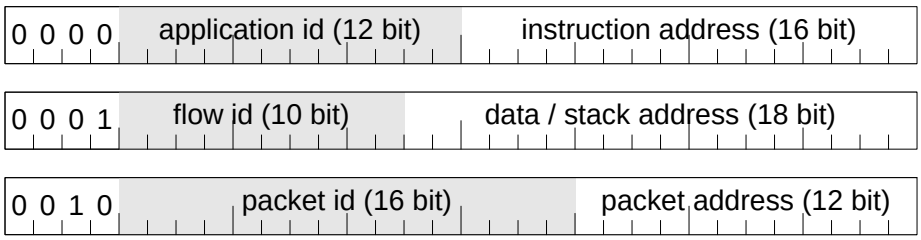


Figure 6.3: Simulation Address Space. The shaded bit fields are adjusted for each application, flow, and packet.

"downwards" from the upper bound of the address segment. Finally, the packet data is limited to 2kB in our simulation and 64k different packets are supported. The packet identifier rolls over after 64k packets, which should not cause problems, because previous packets with the same id will probably have left the router. Using the address space in the above described fashion achieves the desired behavior and does not require any cache flushes or invalidation during the simulation.

6.2.4 Queuing System

The port design shown in Figure 2.4 shows the queue memory as a single memory. In a realistic system, though, there are usually multiple types of memories available. There is fast, but small, off-chip SRAM and slow, but large, off-chip DRAM. There is also a limited amount of on-chip storage on the QCTL. Therefore this memory hierarchy is used to store important, frequently accessed flow data on chip, packet meta-information in SRAM, and the actual packet in DRAM.

The memory layout is shown in Figure 6.4. The on-chip memory contains two per-flow queue data structures. One is used to queue packets before the processing step and the other is used for packets that are ready to be sent out. Each queue requires two pointers (and possibly a counter for the number of queues packets). With 1024 distinct flows, 16kB of on-chip memory are necessary. Additionally, two pointers to the free-lists of SRAM and DRAM data structures are necessary. To avoid memory fragmentation, all data structures are of fixed size. In SRAM, there are packet meta-information data structures, which contain the packet size, the flow classification result, etc., and a pointer to the first DRAM chunk. Packets of the same flow are chained together in SRAM, which corresponds to the per-flow queue. In DRAM, packet data is stored in fixed size chunks, which are chained together for each packet. In the simulation, the SRAM data structure is 64 bytes, which allows

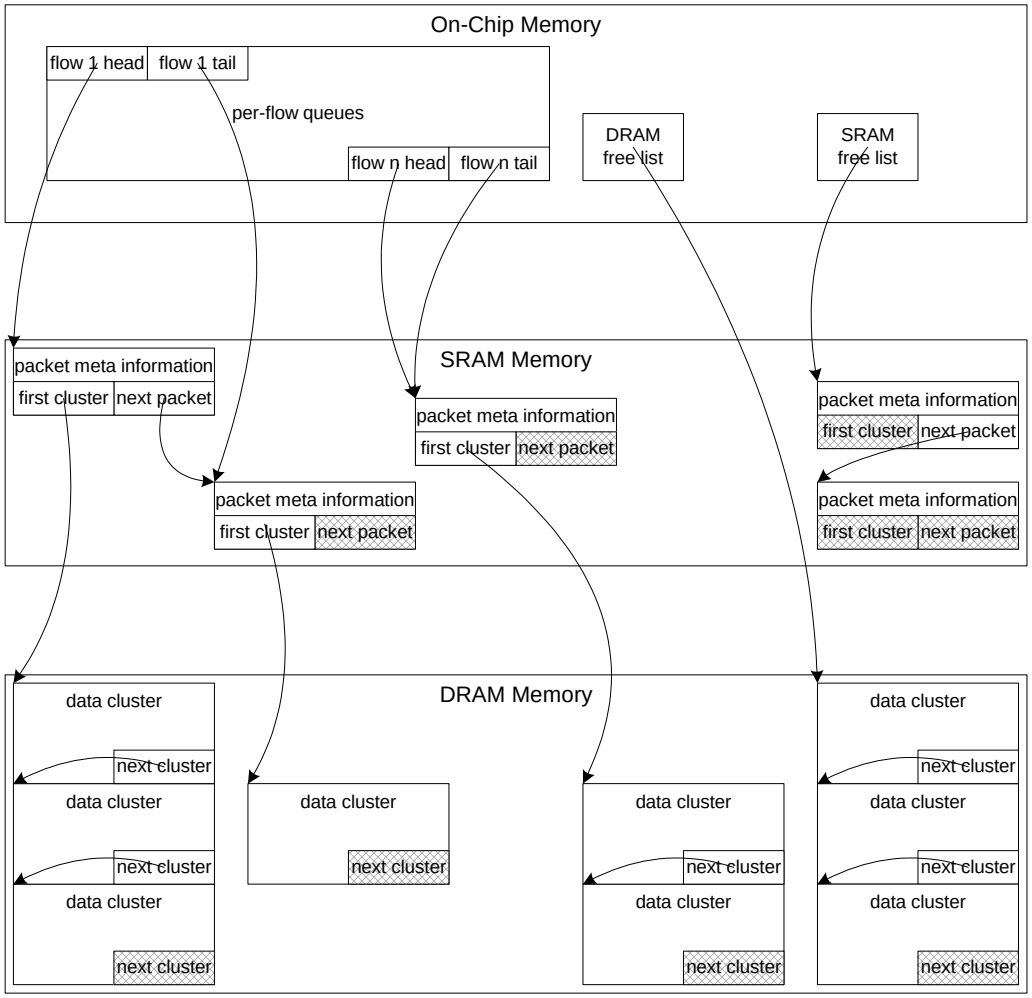


Figure 6.4: Queue Memory Layout. The per-flow queue data structure exists for pre-processing queues and post-processing queues (only one shown).

for 56 bytes of meta-information. The DRAM chunks are 256 bytes, which allows the storage of 252 bytes of packet data.

Accesses to queue memory and the off-chip APC memory are modelled to reflect the typical behavior of SRAM and DRAM memories. Each memory transaction consists of a queuing time to get access to the memory channel, a memory access time, which is technology dependent, and a transmission time, which depends on the memory channel speed and width. This follows the expression given in Equation 4.10. The DRAM access latency (not including the actual data transmission) is assumed to be 60 nanoseconds. For off-chip SRAM this latency is assumed to be 10 nanoseconds. On-chip SRAM is assumed to operate at the clock speed of the processor. As indicated, the simulation does not consider contention for memory banks, which could

limit the ability to interleave memory requests (resulting in a throughput degradation by 20-40%).

6.2.5 Schedulers

There are two schedulers in the queue controller: The processor scheduler, which assigns packets to processors, and the link scheduler, which determines the order of outgoing packets. For the link scheduler, a simple deficit round robin scheme (DRR [SV95]) is implemented. The processor scheduler can implement the scheduling schemes discussed in Chapter 5. In the initial simulation implementation, a simpler round-robin scheduler is used to distribute the workload more evenly over all processors in the system, which leads to more evenly distributed results.

Both schedulers need to maintain data structures for each flow in order to be able to make scheduling decisions. The processor scheduler additionally requires a data structure for each processor. In our implementation, the processor scheduler maintains per-flow information on the size of the first packet in the queue and the requires application. The link scheduler requires the size of the first packet in the queue and the credit of the respective flow. This requires 8kB of on-chip memory for each scheduler.

When a packet is transmitted to the APC or on the link, the respective scheduler needs to be updated on the size of the next packet in the queue. This is done by the queue controller, which reads the meta-information of the next packet in queue memory. Then it informs the scheduler with the PS_FIRST_SIZE and LS_FIRST_SIZE event. This way, the scheduler does not need to maintain a queue of packet sizes for each flow.

6.2.6 Programming Environment

The simulation of packet processing is done by the SimpleScalar tools. However, these tools simulate the entire execution process of a program on a realistic operating system, which includes the preparation of the process context and various other O/S-specific tasks. To not have such processing instructions reproduced in the APC simulation, the following restriction are placed on applications, which allow the simulator to distinguish between application and O/S processing:

- Single Processing Function. All code that should be considered in the simulation has to be placed inside the scope of one function called *process()*. Functions that

```
process.h:

struct flow_state {
    /* here goes any per-flow state data */
};

process.c:

#include "process.h"
extern struct flow_state state;

void process_packet(struct packet *p) {
    /* here goes the processing code for the application */
}
```

Figure 6.5: Application Template. Packet processing functions can be programmed in C using this template.

are necessary for the application have to be placed inside this scope through inlining.

- Single Flow State. All state that has to be preserved among packets of a flow has to be placed inside one structure called *state*.
- Single Packet. A pointer to the packet data structure that needs processing is passed to the processing function. Any modifications to the packet have to be done on this structure. It is not possible to generate or drop packets in the current implementation of the simulator.

With these restrictions, it is possible that the simulator can maintain flow-state and distinguish between flow data and packet data. This is done by scanning the symbol table of the executable to find the location and size of the relevant symbols. Therefore the name of these symbols must not be changed. A template for a processing application is shown in Figure 6.5 (the code for loading and storing packets and flow state is not shown). This is compiled together with some support functions, which load the packet data and flow state before processing and store them after processing. The compiler that is used is *gcc* with the SimpleScalar back-end.

6.2.7 Simulation Summary

The simulation was implemented on a Linux system. The discrete-event simulation is "hand-coded" and the total simulation code (excluding any SimpleScalar tools) consists of about 3500 lines of C code. The simulation speed is mostly bound by the APC simulation, which can perform about 150,000 instruction simulations per second on a 900MHz Pentium III.

In summary, the simulation of the programmable router port considers the most important components of the system, which are the processing engine, memory management, and data flow through the system. The accurate modelling of the processing engine and the consideration of timing issues make it possible to obtain meaningful quantitative results.

6.3 Simulation Results

In this section, simulation results are presented and contrasted to analytic results from Chapter 4. This is helpful to verify the correctness of the analysis, as well as highlight issues that are not captured in the analytic model. In order to obtain comparable results, the workload and system configuration for the simulation and the analytic model has to be the same.

6.3.1 Workload and Configuration

To simplify the comparison, only one application from CommBench is used. The REED application, which performs Reed-Solomon FEC coding, is quite suitable for this purpose. It is computationally intense, which causes the processing engine to be the bottleneck in the system and packets to always be backlogged for processing.

Since the implementation of REED on the simulator is slightly different from the CommBench implementation, there are small differences in the workload characteristics as compared to Chapter 3. The computational complexity is $compl_{REED,sim} = 582$ for encoding, which is considered here. In CommBench, the complexity of $compl_{REED,CB} = 603$ is a bit higher. The load/store frequency in the simulation is $f_{load} + f_{store} = 0.22$, which is slightly lower than in CommBench. Also, smaller caches are considered in the simulation to be able to observe the effects of high miss rates. The miss rates for cache sizes between 128 bytes and 4kB are shown in Figure 6.6.

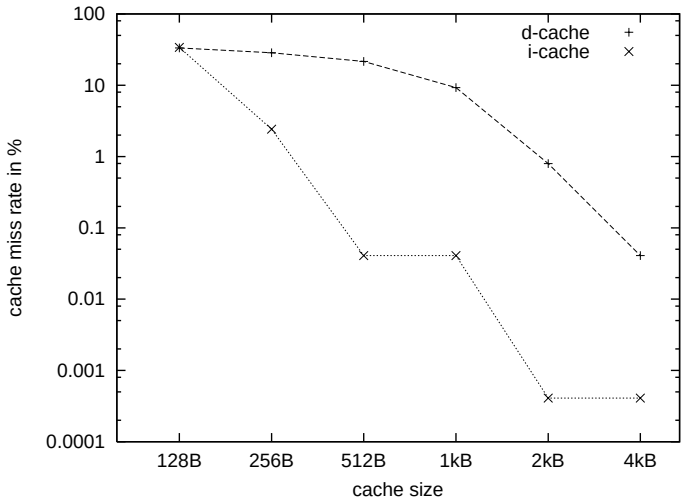


Figure 6.6: Simulation Workload Cache Misses. Instruction and data cache misses for the REED application.

In the simulation, the number of clusters, m, the number of processors, n, the number of threads, t, and the instruction and data cache sizes, c_i and c_d, are varied to observe performance trends. To keep the number of combinations tractable, the instruction cache is always set to the same size as the data cache. The fixed parameters are:

- Processor Clock Speed: $clk_p = 200$MHz.
- Cache Line Size and Associativity: $linesize = 32$ bytes, 2-way associative.
- Memory Channels: $clk_{mem} = 100$MHz, $width_{mem} = 32$ bit.
- I/O Channel: $clk_{IO} = 200$MHz, $width_{IO} = 32$ bit, full-duplex.
- Queue Memory Channel: $clk_{Qmem} = 200$MHz, $width_{Qmem} = 32$ bit.
- Off-Chip Memory Access Times: $\tau_{DRAM} = 60$ns, $\tau_{DRAM} = 10$ns.

The data traffic for the simulation is a sequence of 100 data packets belonging to the same flow. Each packet is 257 bytes long (34 bytes of Ethernet and IP header plus 223 bytes of payload) and requires REED processing. The outgoing packets are 289 bytes including an 32-byte FEC code. The link rate is 2.4Gbps to ensure that packets are available quickly for processing. Due to the complex processing, the maximum throughput of the active router is only a few Mbps (for small configurations).

6.3.2 Comparison

The performance of the analytic model is based on the equations derived in Chapter 4. In particular, the processing power is of interest, which affects the tradeoffs between more processors, more cache, and more memory channels. the following illustrates the derivation of the processing power using the analytic model in the context of the simulation. The configuration is one cluster, one processor, one thread, and 1kB instruction and data caches.

With the workload given, the processor utilization can be determined by determining the overall miss rate p_{miss} and the memory access time τ_{mem}. From Equation 4.14, the miss rate is

$$p_{miss,REED} = 0.00041 + 0.22 \cdot 0.09261 = 0.02075. \tag{6.1}$$

To determine the memory access time, τ_{mem}, it is necessary to consider the load on the memory channel. In the analytic model, the memory channel load is fixed to a particular value. Since we want to compare the results to the simulation, we use the memory channel load of the simulation: $\rho_{mem,sim} = 0.129$. In a configuration with only one processor and one thread per cluster, the queuing delay should of course be $\tau_Q = 0$. Nevertheless, we use the non-zero analytic result for τ_Q to be consistent with the multiprocessor / multithreaded cases. Thus, the memory access time expressed in processor clocks is:

$$\tau_{mem} = 60ns \cdot 100MHz + \left(1 + \frac{(0.129)^2}{2(1-0.129)}\right) \cdot \frac{32byte}{32bit} = 14.1. \tag{6.2}$$

Then, Equation 4.9 gives the total processor utilization:

$$\rho_p = \frac{1}{1 + 14.1 \cdot 0.02075} = \frac{1}{1 + 0.4157} = 0.7739. \tag{6.3}$$

This translates into 77 MIPS of processing power for the analytic model.

The processing power of the simulation for this configuration is also 77 MIPS. A reason for the almost identical processing power results for the simulation and analytic model lies in the negligible difference in parameters. The comparison between the parameters in the analytic model and the simulation are shown in Table 6.1. In configurations where the parameters show larger differences (see below), the processing performance also differs.

To see the accuracy of the analytic model for other, more complex configurations, Table 6.1 also shows the parameters and processing performance for a configuration with $m = 2$, $n = 2$, $t = 2$, and $c_i = c_d = 1$kB. Here, the instruction miss rate and data miss rate differ significantly from the analytic model. This is due to the inaccurate modelling of pollution effects when multiple threads access the same caches. Also there is a slight difference in memory access times. This is due to the packet transmissions over the memory and I/O channel that can delay memory accesses. In the single-thread case this does not occur since the processor is stalled during packet transmissions. With multiple threads, though, one thread might need to access memory while another thread receives or sends a packet over the I/O channel. This can cause significant delays as the I/O-operation cannot be preempted in the simulation. Nevertheless, the analytic processor utilization is only about 3% different from the simulated utilization. This directly translates directly into a 3% error on the processing performance estimation.

The bottom of Table 6.1 shows the results for a configuration with $m = 4$, $n = 4$, $t = 4$, and $c_i = c_d = 1$kB, which is the largest in terms of the number of clusters, processors, and threads that is considered in the simulation. Here, the differences in instruction cache miss rates are enormous. This is due to the fact that the analytic model assumes that different threads cannot share instruction data. In the simulation, however, all threads execute the same instruction code, which is in the same shared address space. As a result, the i-miss rate in the simulation is much lower than in the analytic model. This indicates that the model for cache pollution in Equation 4.15 is not suitable if threads execute the same instruction data. In terms of memory access time, there are also large differences between the analytic model and the simulation. This is again caused by packet transmissions interfering with memory accesses. These two aspects cause the estimated processing power to be off by 23% from the simulation. While this is a significant error, we can see below that this is a particularly bad case of differences in p_{miss} and τ_{mem}.

6.3.3 Error Trends

To show the differences between the analytic model and the simulation over a broader range of configurations, Figure 6.7 shows the processing power of both models for different numbers of clusters, processors, threads, and cache sizes. Also shown is the

Table 6.1: Comparison of Analytic Model and Simulation Results.

1 cluster, 1 processor, 1 thread, 1kB i-cache, 1kB d-cache				
parameter		analytic model	simulation	error
i-miss rate	m_i	0.041%	0.041%	0%
d-miss rate	m_d	9.26%	9.26%	0%
mem. chan. util.	ρ_{mem}	12.9%	12.9%	–
mem. acc. time	τ_{mem}	14.1 cycles	14 cycles	0.5%
proc. util.	ρ_p	77.39%	77.39%	0%
proc. power	IPS	77 MIPS	77 MIPS	0%

2 cluster, 2 processor, 2 thread, 1kB i-cache, 1kB d-cache				
parameter		analytic model	simulation	error
i-miss rate	m_i	0.041%	0.038%	6.9%
d-miss rate	m_d	21.56%	9.26%	132%
mem. chan. util.	ρ_{mem}	30.2%	30.2%	–
mem. acc. time	τ_{mem}	14.5 cycles	16.2 cycles	10.4%
proc. util.	ρ_p	87.6%	90.5%	3.3%
proc. power	IPS	350 MIPS	362 MIPS	3.3%

4 cluster, 4 processor, 4 thread, 1kB i-cache, 1kB d-cache				
parameter		analytic model	simulation	error
i-miss rate	m_i	2.42%	0.046%	5110%
d-miss rate	m_d	28.6%	15.5%	84.7%
mem. chan. util.	ρ_{mem}	79.1%	79.1%	–
mem. acc. time	τ_{mem}	25.9 cycles	60.1 cycles	56.8%
proc. util.	ρ_p	87.7%	71.4%	22.9%
proc. power	IPS	1403 MIPS	1142 MIPS	22.9%

error between the two results (on a logarithmic scale). The baseline configuration is $m = 1$, $n = 1$, $t = 1$, and $c_i = c_d = 1$kB.

For an increasing number of clusters, the analytic processing power estimation is very close to the simulation (maximum error 2%), because the caches and memory channels are replicated and only one thread accesses each. When increasing the number of processors, the error is slightly higher as processors compete for access to the memory channel. The effect of packet transmissions increases the memory access time, which causes τ_{mem} to differ from the analytic model.

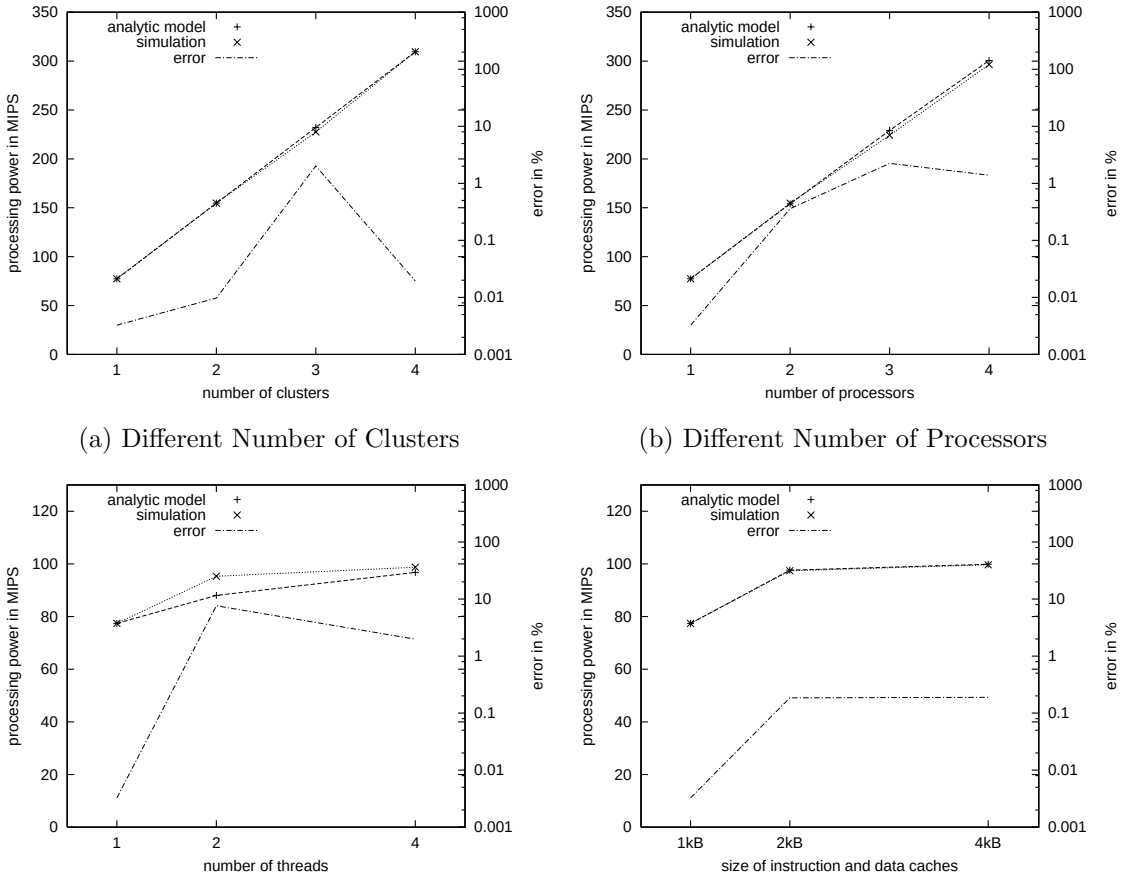
A larger number of threads causes the results to diverge due to differences in cache miss rates. For example, for $t = 2$, the simulation achieves over 7% more processing power than the analytic model predicts, because analytic miss rate is with $p_{miss,analytic} = 0.048$ more than twice as high the simulation with $p_{miss,sim} = 0.021$.

For larger caches, the difference between analysis and simulation are limited as the overall miss rates become very low and have little impact on the overall performance. These trends indicate that the key differences between analysis and simulation lie in the cache miss rates and the memory access times. The following subsections discuss these two points in more detail.

Cache Pollution

As seen in the results above, the analytic model assumes that threads cannot share the data in the cache and therefore effectively only use a fraction of the total cache. However, the instruction cache can well be shared between threads that execute the same application. This sharing is actually desired in the Locality-Aware Predictive scheduling as discussed in Chapter 5. This results in the analytic model being extremely pessimistic in terms of the miss rates that can be achieved, in particular for instruction caches.

Figure 6.8 shows the cache miss rates for different number of threads. Figures 6.8(a) and 6.8(b) show the differences for instruction cache misses. For three and four threads, the analytic model is off by over two orders of magnitude. Less severe is the difference in data cache miss rates as shown in Figures 6.8(c) and 6.8(d). Here, the difference is no more than a factor of 10. As the number of threads increases these errors have a decreasing impact, since the memory accesses can be hidden through multithreading. Therefore, the overall processing power differs only by a few percent, as shown above.



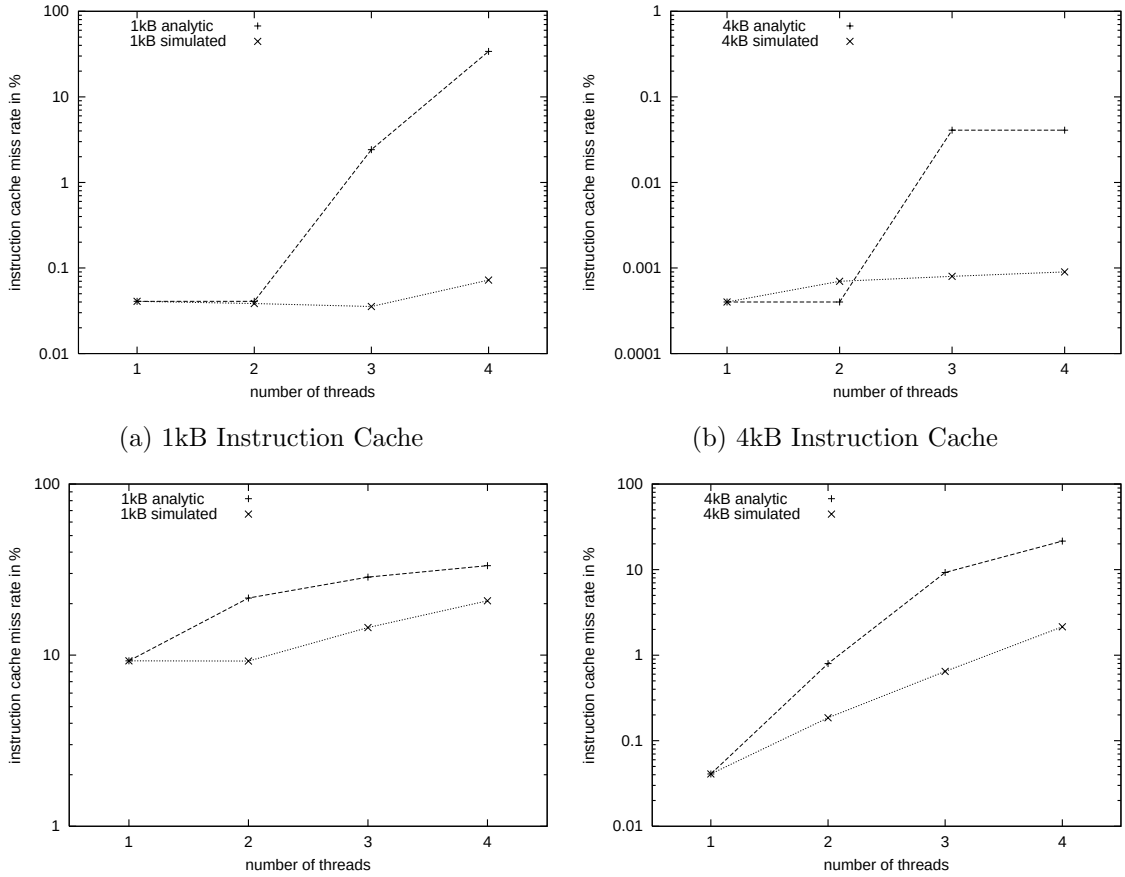
(a) Different Number of Clusters

(b) Different Number of Processors

(c) Different Number of Threads

(d) Different Number of Caches

Figure 6.7: Comparison of Processing Power in Analytic Model and Simulation. The processing power and the error between the analytic model and the simulation results are shown for different configurations. The baseline configuration is 1 cluster, 1 processor, 1 thread, and 1kB instruction and data caches.



(a) 1kB Instruction Cache

(b) 4kB Instruction Cache

(c) 1kB Data Cache

(d) 4kB Data Cache

Figure 6.8: Comparison of Analytic and Simulated Cache Miss Rates. The analytic results assume that cache pollution causes threads to evenly split the available cache. The simulated results show that this assumption is too pessimistic as data can be shared.

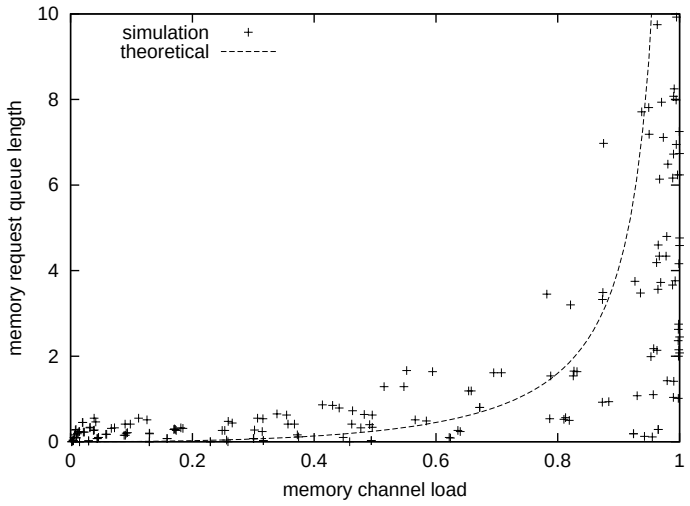


Figure 6.9: Comparison of Analytic and Simulated Memory Channel Queue Length.

The processing characteristics of the workload in the terms of data sharing and cache pollution are clearly not captured well in the analytic model. It can expected that more divers workloads result in better matching parameter estimates.

Memory Access Time

Another cause for differences in the analysis and the simulation is the memory access time. The analytic model does not consider that a packet transmission from the queue controller to a processor locks up the memory channel for some time, which causes memory accesses by other threads and processors to be delayed. Also, the memory access pattern is assumed to be exponentially distributed, which is not necessarily the case. Finally, the case of one single processor and thread is also not captured, because there the memory channel is immediately available for the processor. These points cause the simulated memory access queue length to differ from the M/D/1 approximation as shown in Figure 6.9. Note that the general trend, though, is following the analytic model.

In summary, the comparison between the processing performance analysis and the simulation results shows a good accuracy of the analytic model. In many cases, the difference is only in the order of a few percent. Larger errors are caused by analytic cache miss rates and memory access times that do not match the simulated results. Nevertheless, the analytic model provides a good first-order approximation of the simulated processing power.

6.4 Summary

The simulation of the programmable router port addresses system issues in more detail. In particular, a detailed data and control path is defined and a possible implementation for a queue memory data structures is described. Also, scheduling and programming issues are discussed. The scheduling results are presented and contrasted to analytically obtained results. This comparison shows that the performance model presented in Chapter 4 is accurate within a few percent of the simulation results when cache miss and memory access parameters are similar. The modelling of cache pollution and memory access contention can cause these parameters to differ significantly in the analytic model, which results in processing power estimation that differ by up to 20-30%.

The combination of the analytic performance model, which allows fast design space exploration, and the simulation, which can give more accurate results for selected configurations, is a powerful way for obtaining programmable router configurations with optimal performance characteristics for any workload.

Chapter 7

Summary and Future Work

7.1 Summary

This work presents an extensive discussion of design issues associated with programmable routers. Programmability in the data path of a router allows dynamic deployment of new network protocols and services. This flexibility is achieved by adding general-purpose processing engines to a router port. To achieve the necessary performance, such network processors are designed as embedded parallel multiprocessors. A system design of a programmable router port is discussed and its scalability under consideration of various technology growth trends is shown.

The presented benchmark, CommBench, implements a typical workload for a network processor. The measurements of various characteristics gives a quantitative understanding of application complexity, caching performance, and other processor-architecture related metrics. Together with the analytic performance model, which considers parallel, multithreaded processors, an accurate performance estimate for a broad range of network processor configurations can be derived. This model is used to obtain configurations, which optimally use the area of the embedded system-on-a-chip and achieve maximum processing power per unit of area. From this model, general design tradeoffs for different system components are derived. The simulation of the system provides an additional, more accurate method for obtaining detailed performance results.

The scheduling of processing engines on programmable routers poses a novel challenge that can be addressed by using processing time estimations. By partitioning processors into application groups and scheduling packets such that instruction cache data is reused, lower cache miss rates and higher throughput can be achieved.

Processing time estimations can also be used in the context of reserving processing resources and enforcing fair sharing. Using the proposed scheduling algorithm, bounds on processing delay can be provided.

7.2 Future Work

There are two main areas in which future work can be pursued. One is the networking area, where the questions is how to use a programmable network infrastructure. The other is in the computer architecture area, where more advanced network processors need to be developed to satisfy the need for more processing power.

In the networking area, a particularly interesting question is how to make a programmable network accessible to the user. If the network provides a set of services, how can a user specify that a particular flow should use these services. One approach to solve this is the usage of a programming abstraction similar to "pipes" as they are common in UNIX shells. Such active pipes allow the specification of processing tasks and processing-specific parameters (e.g., processing location restrictions) [KRW01]. The network needs to translate this specification, route the flow, and place processing components accordingly.

With programmable network components becoming deployed throughout the network, interesting new applications can be developed. Programmable routers are particularly suitable to implement applications that not only process packet headers, but the entire payload. One example for such a transcoding application is WWW-document transcoding for thin clients. A mobile, hand-held computer might not have the ability to receive or display large documents (due to a low-bandwidth wireless link and a low-resolution display). A programmable router at the access point could provide a service that automatically reduces images to the appropriate size and possibly converts them to gray-scale. This service would be transparent to the client and the server, which avoids the need for software support on either side.

A key challenge for scenarios, where the network modifies packet payloads, is the issue of end-to-end transport protocols. When TCP was developed, it was assumed that the store-and-forward network would not change the packet (except for data link and network layer headers). With the introduction of store-process-and-forward paradigms, this assumption does not hold true anymore. To still satisfy TCP on the end-system, complex TCP termination or splicing techniques need to be used.

It might be conceivable to develop a different transport layer protocol that supports the notion of processing inside the network.

In the long term, processing inside the network will probably be augmented by storage devices, which have been significantly increasing in data density per cubic inch. With the merging of communication, processing, and storage into a single device, a new class of smart networked servers could be developed. The traditional client-server paradigm will probably be no longer applicable and new peer-to-peer-like mechanisms for communications will be more applicable. The integration of these three key components might help in keeping control over the ever-increasing complexity of modern computer systems.

In the area of network processor design, an important consideration is the use of specialized co-processors for computationally intense networking tasks. The key challenge is to identify the processing steps that are suitable for co-processors, but are also frequently used to effectively utilize such a co-processor. From the results in Chapter 4.4 it is clear that a component should only be added to a network processor if it is used frequently enough to justify the use of silicon real-estate. Candidates for co-processor functions are lookup and classification engines, checksum co-processors, and security co-processors. It might also be conceivable implement specialized hardware for TCP termination to off-load end-systems or to allow payload modifications.

In terms of network processors architectures, the proposed APC design is only one instance of a large class of architectures that exploit flow- and packet-level parallelism. It is conceivable to develop pipelined architectures, which might be more efficient in terms of i-cache locality, but which are more challenging in terms of data movement through the system. In particular with the increasing importance of low power consumption, architectures that reduce data movement might be more suitable. In either case, considering power consumption as the cost function for an analytic performance model will probably be more important than the used chip area.

Finally, with the technology support for on-chip field-programmable gate arrays (FPGAs), it can be considered to provide hardware-programmability as a resource in the network. An FPGA-based implementation of a function can exploit parallelism in a much broader way than a von Neumann processor architecture.

The most exciting challenge for programmable networks lies in the need to continue incorporating results from networking and computer architecture and building a communications and processing infrastructure that provides the functionality and performance for future networks.

References

[AAH+98] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, August 1998.

[AAP+00] George Apostolopoulos, David Aubespin, Vinod Peris, Prashant Pradhan, and Debanjan Saha. Design, implementation and performance of a content-based switch. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[Ada97] Carlisle Adams. The CAST-128 encryption algorithm. RFC 2144, Network Working Group, May 1997.

[Aga92] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPS: The end of the road for conventional microarchitectures. In *Proc. of the 27th Annual International Symposium on Computer Architectures*, pages 248–259, Vancouver, BC, June 2000.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. A processor architecture for multiprocessing. In *Proc. of 17th International Symposium on Computer Architecture*, pages 278–288, Seattle, WA, June 1990.

[ARM99] ARM Ltd. *ARM9E-S - Technical Reference Manual*, December 1999. http://www.arm.com.

[BA97] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Department of Computer Science, University of Wisconsin in Madison, June 1997.

[BO01] Josep M. Blanquer and Banu Ozden. Fair queuing for aggregated multiple links. In *Proc. of ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[BTKM+02] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Jae-Wook Lee, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2), March 2002.

[BZ95] Jon Bennett and Hui Zhang. Worst case fair weighted fair queuing. In *Proc. of IEEE INFOCOM 95*, pages 120–128, Boston, MA, April 1995.

[BZ96] Jon C. R. Bennett and Hui Zhang. Hierarchial packet fair queuing algorithms. In *Proc. of ACM SIGCOMM*, pages 43–56, Palo Alto, CA, August 1996. ACM.

[C-P99] C-Port Corporation. *C-5^{TM} Digital Communications Processor*, 1999. http://www.cportcorp.com/solutions/docs/c5brief.pdf.

[CB02] Patrick Crowley and Jean-Loup Baer. A modelling framework for network processor systems. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[CCF+01] Prashant Chandra, Yang-Hua Chu, Allen Fisher, Jun Gao, Corey Kosak, T. S. Eugene Ng, Peter Steenkiste, Eduardo Takahashi, and Hui Zhang. Darwin: Customizable resource management for value-added network services. *IEEE Network*, 15(1):22–35, January 2001.

[CDMK+99] Andrew T. Campbell, Herman G. De Meer, Michael E. Kounavis, Kazuho Miki, John B. Vincente, and Daniel Villela. A survey of programmable networks. *Computer Communication Review*, 29(2):7–23, April 1999.

[CFBB99] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Workloads for programmable network interfaces. In *IEEE Second Annual Workshop on Workload Characterization*, Austin, TX, October 1999.

[CFBB00] Patrick Crowley, Marc E. Fiuczynski, Jean-Loup Baer, and Brian N. Bershad. Characterizing processor architectures for programmable network interfaces. In *Proc. of 2000 International Conference on Supercomputing*, Santa Fe, NM, May 2000.

[CFFT97] Tom Chaney, Andy Fingerhut, Margaret Flucke, and Jonathan Turner. Design of a gigabit ATM switch. In *Proc. of IEEE INFOCOM 97*, Kobe, Japan, April 1997.

[CGM+01] Kenneth L. Calvert, James Griffioen, Billy Mullins, Amit Sehgal, and Su Wen. Concast: Design and implementation of an active network service. *IEEE Journal on Selected Areas of Communications*, 19(3):404–409, March 2001.

[Chr99] Andrew L. Chraplyvy. High-capacity lightwave transmission experiments. *Bell Labs Technical Journal*, 4(1):43–47, January 1999.

[Cis99] Cisco Systems, Inc. *Cisco 12000 Series Gigabit Switch Routers*, 1999. http://www.cisco.com/warp/public/cc/cisco/mkt/servprod/opt/prodlit/gsr_ov.pdf.

[CK94] Robert F. Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proc. of ACM SIGMETRICS*, Nashville, TN, May 1994.

[Cla88] David D. Clark. The design philosophy of the DARPA internet protocols. In *Proc. of ACM SIGCOMM 98*, Stanford, CA, August 1988.

[Cme93] Robert F. Cmelik. SpixTools instroduction and user's manual. Technical Report TR-93-6, Sun Microsystems Laboratories, Palo Alto, CA, 1993.

[CPU] CPU info center. http://bwrc.eecs.berkeley.edu/CIC/.

[CT98] Yuhua Chen and Jonathan S. Turner. Design of a weighted fair queueing cell scheduler for ATM networks. In *Proc. of IEEE GLOBECOM 98*, Sydney, Australia, November 1998.

[CTW01] Sumi Yunsun Choi, Jonathan S. Turner, and Tilman Wolf. Configuring sessions in programmable networks. In *Proc. of the Twentieth IEEE Conference on Computer Communications (INFOCOM)*, pages 60–66, Anchorage, AK, April 2001.

[CW76] Harold J. Curnow and Brian A. Wichmann. A synthetic benchmark. *The Computer Journal*, 19(1):43–49, February 1976.

[CYB+02] Prashant R. Chandra, Raj Yavatkar, Tony Bock, Mason Cabot, and Philip Mathew. Benchmarking network processors. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[DDPP98] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router Plugins - a modular and extensible software framework for modern high performance integrated services routers. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, September 1998.

[DM92] Murthy Devarakonda and Arup Mukherjee. Issues in implementation of cache-affinity scheduling. In *Proc. of Winter USENIX Conference*, pages 345–357, January 1992.

[DPC+99] Dan Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernhard Plattner. A scalable, high performance active network node. *IEEE Network*, 31(1):8–19, January 1999.

[DRST01] John D. DeHart, William D. Richard, Edward W. Spitznagel, and David E. Taylor. The smart port card: An embedded UNIX processor architecure for network management and active networking. Technical Report WUCS-01-18, Department of Computer Science, Washington University in St. Louis, August 2001.

[EEM] Embedded microprocessor benchmark consortium. http://www.eembc.org.

[EF94] Kjeld Borch Egevang and Paul Francis. The IP network address translator (NAT). RFC 1631, Network Working Group, May 1994.

[EH98] Jan Edler and Mark D. Hill. *Dinero IV Trace-Driven Uniprocessor Cache Simulator*, 1998. http://www.cs.wisc.edu/~markhill/DineroIV/.

[FDW+99] George Frankhauser, Marcel Dasen, Nathalie Weiler, Bernhard Plattner, and Burkhard Stiller. WaveVideo – an integrated approach to adaptive wireless video. *ACM Journal on Mobile Networks and Applications*, 4(4):251–277, December 1999.

[FJ93] Sally Floyd and Van Jacobson. Random early detection (RED) gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[FW02] Mark A. Franklin and Tilman Wolf. A network processor performance and design model with benchmark parametrization. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[GGPY89] Patrick P. Gelsinger, Paolo A. Gargini, Gerhard H. Parker, and Albert Y.C. Yu. Microprocessors circa 2000. *IEEE Spectrum*, 26(10):43–47, October 1989.

[GMC+00] Virginie Galtier, Kevin L. Mills, Yannick Carlinet, Stafan Leigh, and Andrew Rukhin. Expressing meaningful processing requirements among heterogeneous nodes in an active network. In *Proc. of the Second International Workshop on Software and Performance*, Ottawa, Canada, September 2000.

[Gol94] S. Jamaloddin Golestani. A self clocked fair queuing scheme for broadband applications. In *Proc. of IEEE INFOCOM 94*, pages 636–646, Toronto, Canada, June 1994.

[GVC96a] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. A hierarchial cpu scheduler for multimedia operating systems. In *Proc. of the Second USENIX Symp. on Operating System Design and Implementation (OSDI)*, pages 107–121, Seattle, WA, October 1996.

[GVC96b] Pawan Goyal, Harrick M. Vin, and Haichen Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching

networks. In *Proc. of ACM SIGCOMM*, pages 157–168, Palo Alto, CA, August 1996. ACM.

[HBB+99] John J. Hartman, Peter A. Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Oliver Spatscheck, Todd A. Proebsting, Larry L. Peterson, and Andy Bavier. Joust: A platform for liquid software. *IEEE Computer Magazine*, 32(4):50–56, April 1999.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proc. of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[HKN+92] Hiroaki Hirata, Kozo Kimura, Satoshi Nagamine, Yoshiyuki Mochizuki, Akio Nishimura, Yoshimori Nakase, and Teiji Nishizawa. An elementary processor architecture with simultaneous instructions issuing from multiple threads. In *Proc. of 19th International Symposium on Computer Architecture*, pages 136–145, Gold Coast, Australia, May 1992.

[HMS98] Ilija Hadzic, W. S. Marcus, and Jonathan M. Smith. On-the-fly programmable hardware for networks. In *Proc. of IEEE Globecom 98*, Syndey, Australia, November 1998.

[HP95] John L. Hennessy and David A. Patterson. *Computer Architecture – A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, second edition, 1995.

[IBM98] IBM Microelectronics Division. *The PowerPC 405^{TM} Core*, 1998. http://www.chips.ibm.com/products/powerpc/cores/405cr_wp.pdf.

[IBM00] IBM Corp. *IBM Power Network Processors*, 2000. http://www.chips.ibm.com/products/wired/communications/network_processors.html.

[Int00] Intel Corp. *Intel IXP1200 Network Processor*, 2000. http://developer.intel.com/design/network/ixp1200.htm.

[KCD+00] Ralph Keller, Sumi Choi, Dan Decasper, Marcel Dasen, George Frankenhauser, and Bernhard Plattner. An active router architecture for multicast video ditribution. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[KDK+02] Fred Kuhns, John DeHart, Anshul Kantawala, Ralph Keller, John Lockwood, Prashanth Pappu, David Richard, David E. Taylor, Jyoti Parwatikar, Ed Spitznagel, Jon Turner, and Ken Wong. Design of a high performance dynamically extensible router. In *Proc. of DARPA Active Networks Conference and Exhibition*, San Francisco, CA, May 2002.

[KDR+01] Brucek Khailany, William J. Dally, Scott Rixner, Ujval J. Kapasi, Peter Mattson, Jin Namkoong, John D. Owens, Brian Towles, and Andrew Chang. Imagine: Media processing with streams. *IEEE Micro*, 21(2):35–46, March 2001.

[KRW01] Ralph Keller, Jeyashankher Ramamirtham, and Tilman Wolf. Active pipes: Program composition for programmable networks. In *Proc. of the 2001 IEEE Conference on Military Communications (MILCOM)*, McLean, VA, October 2001.

[Lai92] Xuejia Lai. On the design and security of block ciphers. In *ETH Series in Information Processing*, volume 1, Konstanz, Germany, 1992. Hartung-Gorre Verlag.

[Lex00] Lexra Inc. *NetVortex Network Communications System Multiprocessor NPU*, 2000. http://www.lexra.com/products.html.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of IEEE MICRO-30*, pages 330–335, Research Triangle Park, NC, December 1997.

[Luc00] Lucent Technologies Inc. *PayloadPlus^{TM} Fast Pattern Processor*, April 2000. http://www.agere.com/support/non-nda/docs/FPP-ProductBrief.pdf.

[MBC+99] Shashi Merugu, Bobby Bhattacharjee, Youngsu Chae, Matt Sanders, Ken Calvert, and Ellen Zegura. Bowman and canes: Implementation of

an active network. In *Proc. of the 37th Allerton Conference on Communication, Control, and Computing*, pages 147–156, Monticello, IL, September 1999.

[MBZC00] Shashidhar Merugu, Samrat Bhattacharjee, Ellen W. Zegura, and Ken Calvert. Bowman: A node OS for active networks. In *Proc. of IEEE INFOCOM 2000*, Tel Aviv, Israel, March 2000.

[MHN01] Jonathan T. Moore, Michael Hicks, and Scott Nettles. Practical programmable packets. In *Proc. of the Twentieth IEEE Conference on Computer Communications (INFOCOM)*, pages 49–59, Anchorage, AK, April 2001.

[MIP98] MIPS Technologies, Inc. *JADE - Embedded MIPS Processor Core*, 1998. http://www.mips.com/products/Jade1030.pdf.

[MJ93] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proc. of the USENIX Technical Conference*, San Diego, CA, January 1993.

[MLP+01] Sandy Murphy, E. Lewis, R. Puga, R. Watson, and R. Yee. Strong security for active networks. In *Proc. of IEEE OPENARCH 2001*, Anchorage, AK, April 2001.

[MMC00] MMC Networks, Inc. *nP3400*, 2000. http://www.mmcnet.com/.

[MMSH01] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. NetBench: A benchmarking suite for network processors. In *Proc. of International Conference on Computer-Aided Design*, San Jose, CA, November 2001.

[Mog89] Jeffrey C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *USENIX Conference Proceedings*, pages 203–221, Baltimore, MD, June 1989.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[Net] NetBSD Project. *NetBSD release 1.3.1.* http://www.netbsd.org/.

[Pat85] David A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[PEA+96] James Philbin, Jan Edler, Otto J. Anshus, Craig C. Douglas, and Kai Li. Thread scheduling for cache locality. In *Proc. of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[Pet01] Larry Peterson, ed. NodeOS interface specification. Technical report, AN Node OS Working Group, January 2001.

[PG92] Abhay K. Parekh and Robert G. Gallager. A generalized processor sharing approach to flow control: The single node case. In *Proc. of IEEE INFOCOM 92*, pages 915–924, Florence, Italy, May 1992.

[Pso99] Konstantinos Psounis. Active networks: Applications, security, safety, and architectures. *IEEE Communications Surveys*, 2(1), Q1 1999.

[PW02] Prashanth Pappu and Tilman Wolf. Scheduling processing resources in programmable routers. In *Proc. of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, pages 104–112, New York, NY, June 2002.

[QBPK01] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott and Karlin. Scheduling computations on a software-based router. In *Proc. IEEE Joint International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS)*, Cambridge, MA, June 2001. IEEE.

[RF89] T. R. N. Rao and Eiji Fujiwara. *Error-Control Coding for Computer Systems*. Prentice Hall, Englewood Cliffs, NJ, 1989.

[Riv95] Ronald L. Rivest. The RC5 encryption algorithm. In *Lecture Notes in Computer Science*, volume 1008, pages 86–96, Heidelberg, Germany, 1995. Springer Verlag.

[Rob00] Lawrence G. Roberts. Beyond Moore's law: Internet growth trends. *IEEE Computer*, 33(1):117–119, January 2000.

[SBM02] Deepak Suryanarayanan, Gregory T. Byrd, and John Marshall. A methodology and simulator for the study of network processors. In

Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8), Cambridge, MA, February 2002.

[Sem01] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors*, November 2001.

[Sha01] Niraj Shah. Understanding network processors. Technical report, Department of Electrical Engineering and Computer Science at University of California at Berkeley, September 2001. www-cad.eecs.berkeley.edu/ niraj/papers/UnderstandingNPs.pdf.

[Sit00] Sitera Inc. *Prism IQ2000 Network Processor Family*, 2000. http://www.sitera.com/products/iq2000.pdf.

[SJS+99] Beverly Schwartz, Alden W. Jackson, W. Timothy Strayer, Wenyi Zhou, Dennis Rockwell, and Craig Partrodge. Smart Packets for active networks. In *Proc. of IEEE OPENARCH 99*, New York, NY, March 1999.

[SKT96] James D. Salehi, James F. Kurose, and Don Towsley. The effectiveness of affinity-based scheduling in multiprocessor networking. In *Proc. of IEEE Infocom 96*, San Francisco, CA, March 1996.

[SL93] Mark S. Squillante and Edward D. Lazowska. Using processor cache affinity information in shared-memory multiprocessor scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.

[SPS+01] Alex S. Snoeren, Craig Partridge, Luis A. Sanchez, Christine E. Jones, Fabrice Tchakountio, Stephen T. Kent, and W. Timothy Strayer. Hash-based ip traceback. In *Proc. of ACM SIGCOMM 2001*, San Diego, CA, August 2001.

[SSV99] Venkatachary Srinivasan, Subhash Suri, and George Varghese. Packet classification using tuple space search. In *Proc. of ACM SIGCOMM 99*, Cambridge, MA, September 1999.

[Sta95] Standard Performance Evaluation Corporation. *SPEC CPU95 - Version 1.10*, August 1995.

[SV95] M. Shreedhar and George Varghese. Efficient fair queuing using deficit round robin. In *Proc. of ACM SIGCOMM 95*, Cambridge, MA, August 1995.

[SV98] Dimitrios Stiliadis and Anujan Varma. Rate proportional servers: A design methodology for fair queuing algorithms. *IEEE/ACM Trans. on Networking*, 6(2):164–174, April 1998.

[SVSW98] Venkatachary Srinivasan, George Varghese, Subhash Suri, and Marcel Waldvogel. Fast scalable algorithms for level four switching. In *Proc. of ACM SIGCOMM 98*, Vancouver, BC, September 1998.

[TCGK02] Lothar Thiele, Samarjit Chakraborty, Matthias Gries, and Simon Künzli. Design space exploration of network processor architectures. In *Network Processor Workshop in conjunction with Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. of 20th International Symposium on Computer Architecture*, pages 278–288, Santa Margherita Ligure, Italy, June 1995.

[Tra98] Transaction Processing Performance Council. *TPC Benchmark C, Revision 3.4*, 1998.

[T.s99] T.sqware Inc. *TS704 Edge Processor Product Brief*, 1999. http://www.tsqware.com/.

[TSS+97] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[TTG95] Josep Torrellas, Andrew Tucker, and Anoop Gupta. Evaluating the performance of cache-affinity scheduling in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 24:139–151, February 1995.

[TW96] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[VZ91] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proc. of Thirteenth Symposium on Operating Systems Principles*, pages 26–40, Pacific Grove, CA, October 1991.

[Wal91] George K. Wallace. The JPEG still picture compression standard. *Comm. of the ACM*, 34(4):30–44, April 1991.

[WC01] Tilman Wolf and Sumi Y. Choi. Aggregated hierarchical multicast for active networks. In *Proc. of the 2001 IEEE Conference on Military Communications (MILCOM)*, McLean, VA, October 2001.

[Wei84] Reinhold Weicker. Dhrystone: A synthetic systems programming benchmark. *Comm. of the ACM*, 27(10):1013–1030, October 1984.

[WF00] Tilman Wolf and Mark A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, April 2000.

[WF01] Tilman Wolf and Mark A. Franklin. Locality-aware predictive scheduling for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 152–159, Tucson, AZ, November 2001.

[WF02] Tilman Wolf and Mark A. Franklin. Design tradeoffs for embedded network processors. In *Proc. of International Conference on Architecture of Computing Systems (ARCS) (Lecture Notes in Computer Science)*, volume 2299, pages 149–164, Karlsruhe, Germany, April 2002. Springer Verlag.

[WGT98] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *Proc. of IEEE OPENARCH 98*, San Francisco, CA, April 1998.

[WOT+95] Steven C. Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder P. Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methological considerations. In *Proc. of ACM ISCA-22*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[WT00] Tilman Wolf and Jonathan S. Turner. Design issues for high performance active routers. In *Proc. of the International Zurich Seminar on Broadband Communications*, pages 199–205, Zurich, Switzerland, February 2000.

[WT01] Tilman Wolf and Jonathan S. Turner. Design issues for high performance active routers. *IEEE Journal on Selected Areas of Communication*, 19(3):404–409, March 2001.

[Zha95] Hui Zhang. Service disciplines for guaranteed performance service in packet switching networks. *Proc. of the IEEE*, 83(10):1374–96, October 1995.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Trans. on Information Theory*, 23(3):337–342, May 1977.

Vita

Tilman Wolf

Date of Birth May 31, 1973

Place of Birth Marbach am Neckar, Germany

Degrees
D.Sc. Computer Science, Washington Univ., August 2002
M.S. Computer Engineering, Washington Univ., August 2000
M.S. Computer Science, Washington Univ., December 1999
Diplom in Informatik, Universität Stuttgart, July 1998

Experience
IBM T. J. Watson Research Center, Summer 2000 and 2001
Applied Research Lab, Washington Univ., 1998 – 2002
Electronic Radiology Lab, Washington Univ., Summer 1997

Awards and Honors
IBM Research Fellowship, 2001 – 2002
Outstanding Thesis Aware, Universität Stuttgart, 1998
Fulbright Scholarship, 1996 – 1997
Honor Societies: Tau Beta Pi, Eta Kappa Nu

Journal Publications

Tilman Wolf and Jonathan S. Turner. Design issues for high performance active routers. *IEEE Journal on Selected Areas of Communication*, 19(3):404–409, March 2001.

Dan Decasper, Guru Parulkar, Sumi Choi, John DeHart, Tilman Wolf, and Bernhard Plattner. A scalable, high performance active network node. *IEEE Network*, 31(1):8–19, January 1999.

Conference Publications

Prashanth Pappu and Tilman Wolf. Scheduling processing resources in programmable routers. To appear in *Proc. of the Twenty-First IEEE Conference on Computer Communications (INFOCOM)*, pages 104–112, New York, NY, June 2002.

Tilman Wolf and Mark A. Franklin. Design tradeoffs for embedded network processors. In *Proc. of International Conference on Architecture of Computing Systems (ARCS) (Lecture Notes in Computer Science)*, volume 2299, pages 149–164, Karlsruhe, Germany, April 2002.

Tilman Wolf and Mark A. Franklin. Locality-aware predictive scheduling for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 152–159, Tucson, AZ, November 2001.

Tilman Wolf and Sumi Y. Choi. Aggregated hierarchical multicast for active networks. In *Proc. of the 2001 IEEE Conference on Military Communications (MILCOM)*, McLean, VA, October 2001.

Ralph Keller, Jeyashankher Ramamirtham, and Tilman Wolf. Active pipes: program composition for programmable networks. In *Proc. of the 2001 IEEE Conference on Military Communications (MILCOM)*, McLean, VA, October 2001.

Sumi Y. Choi, Jonathan S. Turner, and Tilman Wolf. Configuring sessions in programmable networks. In *Proc. of the Twentieth IEEE Conference on Computer Communications (INFOCOM)*, pages 60–66, Anchorage, AK, April 2001.

Tilman Wolf and Mark A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 154–162, Austin, TX, April 2000.

Tilman Wolf, Dan Decasper, and Christian Tschudin. Tags for high performance active networks. In *Proc. of the Third IEEE Conference on Open Architectures and Network Programming (OPENARCH)*, pages 37–44, Tel Aviv, Israel, March 2000.

Tilman Wolf and Jonathan S. Turner. Design issues for high performance active routers. In *Proc. of the International Zurich Seminar on Broadband Communications*, pages 199–205, Zurich, Switzerland, February 2000.

Tilman Wolf and Dan Decasper. CPU scheduling for active processing using feedback deficit round robin. In *Proc. of the 37th Allerton Conference on Communication, Control, and Computing*, pages 768–769, Monticello, IL, September 1999.

Sumi Y. Choi, Dan Decasper, John DeHart, Ralph Keller, John Lockwood, Jonathan Turner, and Tilman Wolf. Design of a flexible open platform for high performance active networks. In *Proc. of the 37th Allerton Conference on Communication, Control, and Computing*, pages 157–165, Monticello, IL, September 1999.

Workshop, Poster Publications

Tilman Wolf. Network processors - flexibility and performance for next-generation networks. *ACM Computer Communication Review*, 32(1):65, January 2002. (Abstract).

Mark A. Franklin and Tilman Wolf. A network processor performance and design model with benchmark parametrization. *First Network Processor Workshop* in conjunction with *Eighth International Symposium on High Performance Computer Architecture (HPCA-8)*, Cambridge, MA, February 2002.

Tilman Wolf. Network processors - performance and flexibility for next-generation networks. *ACM Annual Conference of the Special Interest Group on Data Communication (SIGCOMM)*, San Diego, CA, August 2001.