APPLYING RECONFIGURABLE COMPUTING TO RECONFIGURABLE NETWORKS

ILIJA HADŽIĆ

A DISSERTATION

in

ELECTRICAL ENGINEERING

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

1999

Jonathan M. Smith
Supervisors

Kenneth R. Laker
Graduate Group Chair

ACKNOWLEDGEMENTS

Although my name is listed on the front page of this thesis it would be unfair not to mention all the people without whom this work would never have become a reality.

Since my very first day at Penn, my advisor Jonathan Smith has provided me with guidance, and encouraged me to pursue my own directions during the course of this research, boldly taking random walks through the unexplored space of science. Four years ago, I was an engineer with a set of technical skills waiting to execute commands sent from the above. Jonathan was the person who showed me that I should not be afraid of going where nobody has been before and helped me become an independent thinker and a scientist.

I am sincerely grateful to my committee members Magda El Zarki, David Farber, Roch Guerin, all of the University of Pennsylvania, and Brendan Traw of Intel Corporation. Their valuable comments and suggestions made me rethink many issues and understand problems in more depth. I thank them for their time and patience in reading my thesis, and for helping me improve the quality of this document.

Among other people who contributed, the person who deserves the most credit is William Marcus of Telcordia Technologies (formerly Bellcore), a great engineer, scientist, colleague and friend, whose technical advice helped me analyze and implement the ideas and concepts presented in this thesis. I am also grateful to Bill for letting me use the SONET boards that he designed and built, which saved a great deal of time in implementing the experimental testbed.

Hilarie Orman and Douglas Maughan from the Defense Advanced Research Projects Agency (DARPA), an organization which funded this research, provided useful comments related to the FPGA security, and encouraged me to continue the investigation in this direction.

I truly enjoyed my work in the Distributed Systems Laboratory (DSL) during the past four years. The DSL provided me with excellent technical conditions for research, and a friendly environment. I wish to thank all the students in the DSL, who in addition to being good colleagues were also great friends. Technical discussions with the members of the lab were a highly valuable source of information.

I would especially like to mention Sanjay Udani who was the first DSL student I met and who introduced me to the lab and helped me adapt to the new environment. Sanjay also contributed to the analysis of FPGA security. Michael Hicks introduced me to the CAML programming language and helped me understand its principles and benefits. I am grateful to him for all the time he spent helping me struggle with this topic. Angelos Keromytis provided advice about many computer security issues, and Steve Muir was a valuable resource for all my Linux kernel questions. I am also grateful to all the anonymous reviewers who contributed comments on my earlier publications related to this thesis.

Besides the technical contributions from my colleagues, this work would not be possible without the moral support from my parents who always encouraged me to go further, and to whom I could always turn when things went wrong. I also wish to mention my aunt, Olga Hadžić, a great mathematician and professor, who in addition to moral support helped me learn mathematics, an essential tool for the analysis presented in this thesis. She always believed that I could, and should, become a scientists even at times when I was unsure.

I have many friends in Yugoslavia, with whom I spent some of the best days of my life. I wish to thank them for all the the fun as well as the moral support.

Finally, and most important, I wish to thank my wife Božana for all her love and understanding. Without her nothing I have done would make any sense.

ABSTRACT

APPLYING RECONFIGURABLE COMPUTING TO RECONFIGURABLE NETWORKS

Ilija Hadžić

Jonathan M. Smith

Considerable research has been recently directed towards building flexible and reconfigurable network infrastructures, which promise to provide better functionality and allow faster evolution in services. Two classes of reconfigurable networks have been investigated: adaptive protocols and programmable (active) networks. An adaptive protocol can modify itself and dynamically optimize its structure in the face of changing network conditions. A second technology, programmable (or active) networks have been proposed as a way of accelerating the deployment and support of new network services. Several software-only prototypes of each technology (adaptive protocols and active networks) have been built. Performance limitations and a tacit assumption that protocol flexibility requires software has generated a considerable amount of skepticism towards these approaches. We make three major contributions in this thesis. First, we demonstrate how results in the field of Reconfigurable Computing can be applied in constructing a reconfigurable network protocol, improving the performance by delegating "bit-intensive" functions to hardware. Second, we extract the safety and security issues discovered in prototyping and generalize them to any architecture using programmable logic. Third, this thesis completes the research in Protocol Boosters, a project in the field of adaptive protocols, which was considered incomplete prior to this work. The methodology we use is evaluation of a proof-of-concept through design and implementation. Our experimental platform, the Programmable Protocol Processing Pipeline (P4) is optimized for network processing and composes a set of field-programmable logic arrays (FPGA) into a processing engine achieving the processing performance of special purpose hardware with the software-like flexibility. Using the P4 and one of the proposed frameworks for building adaptive and programmable protocols, we demonstrate that reconfigurable hardware can be used for building reconfigurable networks. This demonstrates that on-the-fly hardware programmability can be applied in many settings such as line cards, switches and routers.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing and communications research has followed two strong trends. First, the two fields are converging, making computer systems both means for communication (*e.g.*, personal computers used to access information information over the Internet) as well as building blocks for communication systems (*e.g.*, general and specialized computing structures used to implement networking equipment such as routers, switches, network cards, etc.). Second, both computing and communications have been exploring the use of reconfigurable systems. As a result, two new and apparently separate research areas have appeared, exemplified by Reconfigurable Computing[Man97] and Active Networking[TSS$^+$97]. Two classes of reconfigurable networks are being investigated: adaptive networks and programmable networks (of which Active Networks are one example).

Compared to traditionally designed networks, programmable and adaptive networks achieve a higher degree of flexibility and functionality by allowing their behavior to be dynamically modified during operation, either by the user or under control of a policy inside the network. However, the negative performance implications have been a major source of skepticism. Recognizing the problem, some researchers have proposed limiting the concept to the control plane only[LLM96][Sch99] (*i.e.*, out of the data path).

This thesis provides a more ambitious approach to the performance/flexibility tradeoff than restricting the programmability to the control plane. We follow the rationale that for highly parallel, "bit-intensive" processing, special purpose hardware performs better than software executed on a general purpose processor. Demands for high performance processing in the data plane favor hardware implementations, while the programmability and flexibility requirements favor software implementation. To address these opposing demands we use reconfigurable computing to implement reconfigurable networks.

SRAM based Field Programmable Logic Arrays (FPGAs) implement algorithms in hardware with the ability to be reprogrammed at run-time. This offers the potential of achieving hardware speeds while keeping software-like flexibility. This has been exploited to speed up algorithms without the loss of generality of computing structures. The FPGA devices have proved useful in specialized bit-intensive tasks, mainly due to the parallelism exploitable by the hardware. The existence of such tasks in network processing represents an opportunity for FPGA devices to improve system performance while maintaining the flexibility necessary for adaptive and programmable networks.

1.1 Programmable (Active) Networks

Network functionality can be abstracted as a collection of services offered to the user where network evolution can be viewed as a process of adding new, and obsoleting legacy, services. In the past decade, networking community has made a significant research effort in defining and deploying new services in various types of networks.

A consequence of the traditional approach which implements all services using a common interoperability layer in the protocol stack (*e.g.*, IP on the Internet[Pos81a]) is that it makes network evolution slow as modification of this interoperability layer involves a painful migration process. An example is the transition from IPv4 to IPv6[DH98], which has still not occurred, despite the attempts to standardize the strategy for handling the incompatibilities between the two versions[GN96]. This approach thus represents a major obstacle to efficiently adding new services which are not well supported by IPv4, and thus require modifications to the network infrastructure. For example, real-time video and voice services require a certain amount of Quality of Service (QoS), while IP supports only best-effort packet delivery.

Attempts to provide a "universal" network infrastructure, such as ISDN[DNSS94] or ATM[Min89] which could support all current and prospective services did not prove to be successful either. These technologies ended up being used where they were not intended. For example ATM is mostly used for high speed backbone IP networks[JW97] with no QoS or constant bit rate telephony, while ISDN is mostly used for residential Internet access[Tan96].

To overcome the problem of slow network evolution, embedding network programmability has been proposed. A basic idea is that if the programming interface, instead of a layer in the protocol stack, were specified as the interoperability layer, it would be possible to implement and rapidly deploy new protocols and services. Rather than having to agree completely on how to transfer packets, network implementors would have to agree on how to define and distribute network services. While the concept allows more flexibility, by putting the network functionality in the hands of a user, the increased processing burden required of the network has potentially large negative performance and security implications.

1.2 Reconfigurable Computing

In industry, programmable logic has seen wide application in rapid prototyping and low-volume production. In the prototyping stage, programmable logic allows engineers to shorten the design cycle and test the circuit in a realistic run-time environment rather than a potentially inaccurate simulation. Further, in the early stages of production, it allows marketing the product without the time and consequent financial risk associated with an application specific integrated circuit (ASIC) design.

On the other hand, the research community has seen run-time reconfiguration as a means for creating new concepts in computing, namely reusing the same silicon die for different designs that can be chosen during operation. The ultimate goal is to combine the performance of dedicated hardware with the flexibility of software systems. Xilinx was the first programmable logic manufacturer to recognize the potential of reconfigurable computing by designing an FPGA device specifically optimized for this application[Xil97].

Since hardware is inherently parallel, algorithms that can exploit this parallelism can achieve remarkable performance on systems augmented with run-time reconfigurable logic.

Significant performance improvements compared to the use of general purpose systems have already been achieved in the field of digital signal processing, image processing, DNA pattern matching, data encryption, etc. Assuming an application that utilizes many bit-intensive tasks, but not all at the time, such as active networking, reconfigurable hardware can provide an attractive cost-performance tradeoff, as the algorithms can be implemented in hardware with the ability to recycle the same physical component when the set of required functions changes. Namely, if the system utilizes a set of functions implemented in hardware for improved performance, and only a smaller subset is needed at the time, it may be more economical to use programmable logic than to build the hardware for each available function.

As we will discuss in Section 2.5 certain classes of protocol processing algorithms can efficiently exploit the parallelism offered by this hardware. This enables the implementation of performance critical parts of the protocol stack in hardware, without a major loss to the flexibility required for programmable networks.

1.3 Summary of Contributions

The major problem addressed in the first part of this work is the performance gap between processing power and line rates, which increases when making the network infrastructure flexible. We show that by augmenting a network infrastructure with programmable hardware, performance improvement is possible at a small or even negligible price in flexibility. We follow the rationale that the convergence of the computing and communications fields applies to reconfigurable computing and reconfigurable networking. The result is a programmable network infrastructure where the end-user is allowed to define and modify hardware portions of the protocol stack.

We use two models as the framework for demonstrating the validity of our concepts. First is the Protocol Boosters model proposed by Feldmeier[FMS+98][MHMS98] which provides an adaptive network infrastructure through run-time insertion and removal of modules on an as-needed basis. The second model is an extension of Feldmeier's original concept which allows the user to define and deploy modules making the network fully programmable. This extension represents a link between the Protocol Boosters model which focuses on adaptivity, and an Active Network, which focuses on programmability.

To experimentally verify the validity of using reconfigurable hardware for building adaptive and programmable networks, we have designed and implemented a platform called the Programmable Protocol Processing Pipeline[HS97] which composes a set of Altera Flex10K[Alt98d] devices into a run-time reconfigurable processing engine. Although we focused on implementing the testbed based on Protocol Boosters and our extension of the model, the P4 architecture can be used as a general platform whenever a networking application requires the combination of hardware performance and flexibility.

The Protocol Boosters model which we use in our analysis and experiments has been designed as a methodology for implementing adaptive protocols through their incremental construction out of mechanisms under the control of a policy. However, the inability to provide a convincing example of a policy has hindered claims that Protocol Boosters are a viable means of dynamically adapting protocols. In this thesis we provide the first design and implementation of such a policy module and demonstrate the benefits of adaptation on an example, that is dynamic instantiation and removal of a Forward Error Correction (FEC) mechanism.

In our programming model, the user is allowed to define and deploy processing modules which are implemented in hardware as FPGA configurations. In the second part of the thesis we show that this unfortunately opens another avenue for malicious users to create hardware analogous of computer viruses. We identified and analyzed this problem in the context of programmable networks. However, our results are not limited to networking and apply to any system that utilizes programmable logic. The analysis leads to a general security threat model of run-time reconfigurable computing systems. This result opens up a new discipline in the Reconfigurable Computing research which is another major contribution of our work.

To summarize our claims, we list the scientific and technical contributions in order of importance:

1. Demonstration that reconfigurable computing can be applied to realize a reconfigurable networking infrastructure.

2. Development of a general security threat model for systems utilizing run-time reconfigurable hardware, possible methods of protection, and opening a new discipline in Reconfigurable Computing: Reconfigurable Computing Security.

3. Design and implementation of a run-time reconfigurable platform, the Programmable Protocol Processing Pipeline (P4), specifically optimized for network processing. The P4 accelerates bit-intensive tasks in network protocols while keeping software-like flexibility.

4. The first complete example of a Protocol Booster with the implementation of both mechanism and policy module.

5. Extension of the Protocol Boosters model to allow end-user programmability.

6. First operational Active Network infrastructure (hardware and software), called Field Programmable Logic Active Network Environment (F-PLANE), which provides the programmability of hardware portions of the protocol stack.

1.4 Methodology

The methodology we use to demonstrate the validity of our claims, is a proof-of-concept through experimenting with *artifacts* (*i.e.*, hardware and software systems) built for the purpose of demonstrating that the concept is feasible and that it achieves its objectives [BC94].

We have built the Programmable Protocol Processing Pipeline (P4) to provide a hardware support for our concept of extending network embedded programmability to hardware and implemented a software system that runs on a network consisting of hosts, switches and P4 boards. The software manages P4 boards distributed over a network, synchronizes their operation through a specially designed signalling protocol, and allows the end-user to control the operation of the system. We call our testbed the Field Programmable Logic Active Network Environment (F-PLANE).

In the F-PLANE testbed we evaluate the performance of a dynamically instantiated Forward Error Correction (FEC). We have chosen the FEC as a simple and well defined function that is easy to understand yet still useful in demonstrating the concept. We experimentally show the benefits of adaptation when the FEC module is functioning as a booster instantiated on an as-needed basis and the advantages of implementing the function in hardware. We further show the mechanism for deploying the function when it is defined by the user (*i.e.*, Extended Protocol Boosters Model) and project our results to a general case.

The focus of our discussion is on the benefits of *adaptation* and *programmability*, not the FEC coding technique. We are therefore not concerned about behavior of any particular error correcting code, or error models, which have been extensively studied in work by others [SJ89],[HL87],[SM92],[HW98]. Rather, we focus our attention on offering network programmability and adaptivity in hardware.

1.5 Outline

The remaining chapters of this thesis are organized as follows. Chapter 2 presents a summary of related work and compares it to our research. Chapter 3 describes the architecture of the Programmable Protocol Processing Pipeline (P4), a platform we constructed to experimentally validate our approach. A case study is the focus of the Chapter 4. We use a simple protocol processing function to illustrate the benefits and challenges of an adaptive protocol. In Chapter 5, we extend the model to provide full user programmability. Chapter 6 discusses a general security threat to systems that utilize the programmable hardware. We start by discussing security issues specific to our concept, and extend the discussion to the general case of reconfigurable computing. We conclude the thesis in Chapter 7 by reviewing our contributions and outlining future directions of the research. Additional details on some topics addressed in the thesis are provided as Appendices A and B.

Chapter 2

Background and Related Work

In this chapter we provide the background necessary for reading and understanding the discussion, analysis and experiments presented in the remaining chapters. We assume that the reader has a general knowledge of networking, digital systems and computer hardware.

We explain concepts of adaptive protocols, network embedded programmability and reconfigurable hardware and summarize the related work in these fields.

2.1 Adaptive and Programmable Networks

A general observation is that systems designed and optimized for specific conditions perform better than general purpose systems. The trade-off between generality and in-context performance has been an object of scientific study in many fields.

Adaptive systems automatically optimize their characteristics during the operation based on current conditions. Adaptation can be achieved either by modifying parameters of the system or by qualitatively changing the system functionality. *Programmable* systems, on the other hand, are designed to allow the user to customize the system behavior through a well defined user interface or a programming model.

Traditionally, a computer network has been viewed as a static resource, and adaptation (if any) is achieved through a limited set of parameters. For example, TCP[Pos81b] adapts its congestion window size based on current traffic conditions using timeouts and duplicate acknowledgements to detect the packet loss[Jac88]. While this scheme works well if all packet losses are due to network congestion, it is simply the wrong strategy for a noisy environment where packet losses due to bit errors dominate. The TCP example illustrates how an adaptation through parameters works well for one class of network conditions (variable traffic conditions) while the other class (variable bit error) is misaddressed. Moreover, TCP has no means of addressing bit errors other than retransmission and qualitative changes to protocol stack are necessary to solve this problem. Many researchers have addressed the problem of improving TCP performance in a noisy environment by adding extra functionality into the protocol stack with the goal of optimizing the protocol for a new environment[BPSK96]. The simple addition of new functions does not make the protocol adaptive in the sense of being able to qualitatively modify its behavior. A fully adaptive protocol must be able to dynamically modify its functionality during operation without disrupting the ongoing communication. In the context of adaptive protocols, means of improving the TCP performance in different environments can be viewed as *mechanisms*, while the decision on when and which mechanism to use is under control of a *policy*.

The quantitative measure of efficiency of an adaptive protocol is the performance improvement compared to a traditionally designed protocol, which depends on three factors. First, the overhead introduced by a mechanism must be low enough that it improves performance for a significant operating range of the protocol. Second, and demonstrably more important, policy modules must correctly identify opportunities for deployment of mechanisms. Third, the time required to react to the opportunity (*i.e.*, deploy the mechanism) must be short enough to ensure that the opportunity does not disappear. This third factor, for example, might suggest that, for short fluctuations in network conditions no action be taken to avoid "thrashing" of the protocol graph.

Programmable networks modify their functionality under the control of a user. While the main goal of adaptive networks is to cover wider set of network conditions under which the system can operate with acceptable performance, or alternatively improve the overall system performance if the range of varying network conditions is large, programmable networks improve the system flexibility allowing rapid deployment of new services, customizing the protocol to the application, using the network as a computing resource, and testing new concepts. Alternatively, a programmable network can be viewed as an adaptive network in which defining a mechanism and making policy decision has been delegated to the user. Given their different objectives, adaptive and programmable networks can be viewed as orthogonal concepts.

Network programmability has been aggressively researched through a DARPA sponsored Active Networking initiative[TSS+97]. Two fundamentally different concepts of building active networks have been defined. The first, involves downloading services into the network elements and later referring to them using a field in the packet header. The second concept views packets as a combination of a program interpreted at the active network node and the payload data. In this approach every packet carries a small chunk of executable code which defines how the packet should be handled at the node.

Network programmability offers new functionality, but on the other hand exposes network resources to the user. Hence, safety and security are major issues in programmable networks. Programming model must be designed to offer sufficient expressibility to the user so that the useful services can be defined, but on the other hand it must be restricted to prevent malicious users from attacking network resources. Finally, the overhead of adding programmability to the system must be sufficiently low to enable system operation with acceptable performance.

2.2 Related Work in Adaptive and Programmable Networks

2.2.1 X-Kernel

X-Kernel[HP91] is a protocol implementation framework with low per-protocol overhead, designed for experimentation in network protocol design. Previous to X-Kernel, complex functional elements (protocol layers) constituted simple *protocol graphs* with very little or no dynamics in functional behavior[OP92]. The approach taken in the X-Kernel calls for simple functional elements to be composed into complex protocol graphs, where a message does not necessarily traverse all nodes of the graph. Adaptation is achieved through the introduction of "virtual protocols" which direct the message to only the necessary portions of the protocol graph. For example, if the message is destined to the local network, all IP processing may be bypassed.

2.2.2 Protocol Boosters

Protocol Boosters[FMS+98][MHMS98] model is representative of an adaptive protocol that dynamically optimizes the protocol stack to current network conditions. The goal of the boosters work was to improve end-to-end performance by appropriate runtime selection of functions.

In this model, the system consists of a base protocol and a set of mechanism modules which represent building blocks. The adaptation is achieved by run-time insertion and removal of available mechanism modules. The base protocol together with a set of mechanism modules in the protocol stack represent an instantiation of an adaptive protocol. Clearly, not all available mechanisms must be in the protocol stack concurrently, and the set of inserted mechanisms is a subset of those available. The result is a light-weight protocol capable of modifying itself in the face of changes in network conditions. Insertion and removal of mechanism modules are under a control of an associated policy module.

Compared to the X-Kernel approach, Protocol Boosters build the protocol graph dynamically at run-time, while the X-Kernel assumes a protocol graph built at compile time, with virtual protocols allowing bypassing of unnecessary portions of the protocol graph.

2.2.3 XBind

XBind[LBL95][LLM96] is a software system developed at Columbia University that uses a distributed object technology to abstract the creation of new multimedia services on the top of an existing QoS-supporting network infrastructure. In this project network programmability has been implemented in the control plane, while the data plane remains a classical network. The user programs that implement new services are mapped to QoS requirements and signalling supported by the underlying network infrastructure. The user has the ability to create and deploy new services without a loss in performance from adding programmability support in the data plane.

2.2.4 ANTS

The ANTS project at MIT[WGT98] is representative of an Active Network which achieves programmability by allowing users to download the code into the network elements. Data packets carry the identifiers in their headers which refer to the code that needs to be executed to process the packet.

If the appropriate code does not exist at the node, it is requested from the previous hop where the code is presumably cached since the packet had just visited it. The next packet requesting the same code and following the same path will find the appropriate code already cached. Hence, the code propagation mechanism is triggered by the packet arrival and new code is spread as packets traverse the network topology.

2.2.5 Smart Packets

The Smart Packets project[Sch99] at BBN Networks takes an approach orthogonal to that of ANTS. Network programs are carried in packets and interpreted either at each hop or at the endpoint. A special purpose programming language has been designed to provide a balance between functionality and security of the system.

Performance issues that arise from high processing burden on the network nodes have been addressed by restricting the programmability to network management and monitoring only. They argue that despite the faster growth in network bandwidth than growth in processing power, the amount of traffic in the management plane has not increased, so that increasing processing power on the network nodes can be used to provide new functionality.

2.2.6 Switchware

The Switchware project[Ale98a] combines concepts of downloadable services and packet-programs into a unified model of active network. The header field of an active packet carries a program implemented in a domain-specific programming language, PLAN[HKM+98]. The CAML based active packets are also supported[Ale98a]. A PLAN packet is evaluated on each active node; these can be located at every hop or after multiple hops. Similarly to the language used in the Smart Packets project, PLAN is a tradeoff between limitations imposed by security requirements and the expressibility necessary for writing useful packet-programs.

PLAN packets can invoke node-resident services to achieve functionality not offered by the language itself. Services may be deployed by the user and the security is achieved by using a type safe language and authentication methods from public key cryptography[Ale98].

2.2.7 Hardware for Active Networks

Many research groups have either neglected the performance issues of flexible network infrastructures, or restricted the concept to control planes. Very few researchers have attempted to provide the specialized hardware for active networks. Our first prototype of the P4 described in [HS97] was a pioneering effort that opened a new avenue of research in the active networking community.

Two other research groups are new following this direction, recognizing the benefit of using hardware specialized for programmable networks. The first group at Washington University in cooperation with ETH Zurich proposed an architecture which combined a general purpose processor with an FPGA at every input port of an ATM switch and provided a distributed processing engine with a specialized operating system controlling the hardware[Dec99].

The second group at the Virginia Polytechnic, assumed the architecture very similar to ours and discussed dynamic hardware-software co-design in the context of active networks [LJMA99]. They proposed a concept of partitioning the active network program into software and hardware modules and scheduling the execution on general purpose CPU and FPGA processing engine respectively.

2.2.8 Relationship to our Work

Our research was initiated to address the performance issues of Protocol Boosters implemented in software for Linux[MMR98] and Free-BSD[MCS97] operating systems. The major finding of the previous work was that the dominant performance cost is not in overhead introduced by adding boosters support to existing protocol implementation, but in data processing overhead from the deployed boosters. We designed a board, the Programmable

Protocol Processing Pipeline (P4)[HS97], whose architecture was suited for the Protocol Boosters model. With P4 we enabled applying the same methods from the boosters work to hardware portions of the protocol stack.

The P4 allowed us to build the adaptive network infrastructure, with some functions implemented in programmable logic. Similar approach could be taken in building a programmable network where both hardware and software functionality is exposed to the user. Hence, we believed that the concepts learned from building the P4 board were not limited to Protocol Boosters and can be applied in any type of Active Network or any other work related to network embedded programmability. Following this rationale, we provided the logical extension of Protocol Boosters model allowing the end-user to program the modules creating a fully programmable network infrastructure extending the focus of Protocol Boosters model from *adaptation* to *adaptation and programmability*.

Our work, therefore, addresses the problem of reducing performance costs in programmable and adaptive networks by delegating portions of processing burden to special purpose hardware. While XBind and Smart Packets address the performance problems through restricting the programmability to control or network management plane we attempt to provide the programmability in the data plane with the ability to achieve high performance. The cost of our approach is in the need for special hardware (*i.e.,* programmable logic) and limited complexity (constrained by available gates) that can be embedded in the programmable logic.

Our work fills in a portion of the design space that has not been investigated yet. Thus, our results represent an extension of results achieved by other research groups.

Table 2.1 on page 23 summarizes the discussion presented in this section. The last two rows (Protocol Boosters on P4 and Extended Protocol Boosters on P4) are the models we used for implementation and experiments. Successful addition of full programmability to the Protocol Boosters model and addressing major concerns related to programmable network infrastructure demonstrates the validity of our claim that network embedded programmability is not limited to software systems. Although we used a particular model for implementation, we will discuss how the results and lessons learned from our research can be applied to *any* other model of programmable network infrastructure.

2.3 Programmable Logic and Reconfigurable Computing

Programmable logic has been widely used for prototyping and low volume production in early stages of product marketing. In general, the architecture of these devices consists of logic blocks, routing resources, I/O blocks and a configuration memory where the user's design is stored.

Logic blocks consist of lookup tables for implementing combinatorial logic and some small amount of sequential logic (*i.e.*, flip-flops). Logic blocks can be internally connected and locally grouped into various structures such as combinatorial circuits, counters, state machines, etc. Local and internal interconnects usually reach only within the logic block or a group of adjacent logic blocks and connect via multiplexers whose selection inputs are controlled by bits in the configuration memory.

Logic blocks or groups of logic blocks can be interconnected using routing resources that span the entire chip or some large portions of the chip in both vertical and horizontal directions. Connections between logic blocks and routing resources are achieved via a pass gate which is also controlled by the configuration memory. Some devices utilize crosspoint switches at the intersections of vertical and horizontal signal lines for improved routability.

Routing resources connect to I/O pads on the chip via an I/O block which may contain some limited programmability. Programmability of the I/O block usually includes selecting whether the pad is an input, output, bidirectional, open-drain etc., and whether the signal is fed directly or synchronized to the global clock by a flip-flop.

In addition to general purpose routing resources, many programmable logic devices utilize global routing resources which directly reach every logic block on the chip. Global routing resources are mostly used for clock distribution and distribution of global signals such as reset.

Programming the device is achieved by setting bits in the configuration memory which control logic blocks and their interconnections. The typical engineering cycle involves creating a design using schematic entry or a hardware description language such as VHDL[IEE93] or Verilog[IEE95] and compiling it into the configuration using the compiler provided by the device manufacturer. The device is configured by loading its configuration memory.

The architecture of a programmable device is usually a result of a trade-off between routability and delay predictability. Unpredictable delays make the design difficult, because the engineer is unable to estimate the performance of a design in advance and the only way to find out the maximum clock rate at which the device would operate is to compile the design and evaluate its behavior.

The major source of delays is in the routing resources. With the increased routing complexity, register to register delay (which determines the maximum clock rate) variation increases especially if the crosspoint switches are involved in routing, resulting in unpredictable circuit performance. On the other hand, simple routing provides predictable delays at the expense of poor device utilization because of the possibility of not finding a route even if not all resources are used.

Configuration memory can be implemented either as a one-time programmable memory (PROM) non-volatile reprogrammable memory (EPROM, EEPROM or FLASH) or as a fully alterable memory (SRAM). In the first case, devices are programmed once during the design phase and used in the circuit as fixed logic. The second option allows device reconfiguration, which can be used for field upgrades without the need to replace the entire board. The ability to do in-system programmability (ISP) makes these devices popular in systems which are expected to be frequently upgraded. Finally, implementing the configuration memory in SRAM allows run-time reconfigurability, the ability to reconfigure the device while the rest of the system is operating normally. Run-time reconfigurability differs fundamentally from in-system programmability since the circuit may be reconfigured several times while executing a task, making the model similar to that of software systems: downloading different configurations is analogous to executing different programs. Some programmable logic devices support the run-time reconfigurability concept on a device level, allowing partial reconfiguration without disrupting the operation of other parts of the chip[Xil97].

Run-time reconfigurability of programmable logic is a technical foundation for our work. We exploit this feature to create a system with both the flexibility inherent to software systems and the potential of achieving hardware speeds. Evidence from Computer Scientists working in Reconfigurable Computing suggests that this approach is both feasible and attractive in selected applications.

2.4 Related Work in Reconfigurable Computing

2.4.1 SPLASH Host Accelerator Board

A representative example of related work was done at the Institute for Defense Analyses Supercomputing Research Center in Bowie MD, and combined a pool of 32 devices from the Xilinx XC4000 family[Xil99] into a system called SPLASH, which was attached to a computer workstation[ABD92]. Data patterns were compiled into matching programs for the SPLASH, which was then able to operate in parallel, at full hardware speeds, on a data stream. The architecture assumed a dedicated task and that all available resources would be devoted to that task until complete. The application consisted of a program executed on a workstation and a set of configurations used to program the SPLASH. The SPLASH board functioned as an accelerator that improved the workstation performance by mapping highly parallel algorithms to hardware. For certain classes of algorithms, performance of such systems was comparable to that of supercomputers[Gok91].

2.4.2 XPuter Architecture

Research at the at the University of Kaiserslautern, Germany, has focused on defining a general purpose architecture suitable for efficiently mapping algorithms onto programmable logic.

The result was the XPuter paradigm introduced by Hartenstein[HHW90] as a data driven architecture with a programmable ALU. Since then, the work has been done in implementation, identifying applications suitable for this model and providing operating system support[KHN97]. XPuter architectures use a data sequencer to feed the reconfigurable ALU. In contrast to classical computing architectures, an XPuter has no instruction sequencer and programs are thus purely data driven. The data sequencer scans the data memory using a pattern defined by the application. Data patterns are processed by the reconfigurable ALU and written back into the memory or made available to the host. XPuter architectures can be efficiently implemented with FPGAs and can outperform classical computing architectures in highly parallel applications such as signal and image processing.

Compared to the SPLASH model, in which a programming model is at its lowest level and mapping algorithms relies on the skill and experience of a hardware engineer, XPuter provided a programming model at a higher level of abstraction making it more accessible to programmers.

2.4.3 PipeRench: A Different View of Pipelining

The PipeRench project at Carnegie Mellon University addresses programmable hardware virtualization for pipelined applications[CWG+98]. If application processing can be broken into sequential stages, one can either implement the full pipeline (the traditional approach) or use reconfigurable logic to multiplex stages of the pipeline during the execution. For example, two stages of the pipeline can be active while others are cached, waiting to be downloaded when needed. The pipeline speedup depends on the number of stages that can be active concurrently, which in turn depends on the device size. The research goal is to hide the physical size of the device (except for the performance penalty paid if the device is too small), analogous to virtual memory in general purpose computing systems.

This research provides an interesting view of pipelining enabled by run-time reconfigurability. Traditionally, pipelining is used to speed up the throughput of the processing unit. In PipeRench project, pipelining is used to provide a tradeoff between the size and throughput. At one extreme, the device configuration is changed at every stage in the pipeline processing, and the task is executed on a very small device with no parallelism. On the other hand, the task can be fully pipelined, occupying a large area on the chip. PepeRench covers the full spectrum between these two extremes.

Because of the need to support for rapid reconfiguration in each stage of the pipeline, this approach requires a special device feature: configuration caching. The team at CMU is building a programmable logic device with multiple banks of configuration memory and a mechanism to activate each cached configuration on a per cycle basis[TSE98].

2.4.4 Resource Management in Partially Reconfigurable Circuits

The Reconfigurable Architecture Group (RAGE) at the University of Glasgow, Scotland is researching the resource management issues in partially reconfigurable systems[BDH+97]. They used formal methods of Partial Evaluation[JPS85][JGB+90] to specialize configurations during run-time, prior to downloading them into the device. The result of the research was operating system support for dynamic partial hardware reconfiguration. The system utilized a commercially available Xilinx XC6126 PCI board and improved the application performance by delegating functions to hardware. Choosing which portion of the chip to reconfigure and what function to delegate to hardware was the responsibility of an operating system. The authors demonstrated their concept on a Postscript rendering example.

2.4.5 Relationship to our Work

Many platforms for Reconfigurable Computing community have targeted *computing* applications. Researchers tried to stimulate the adoption of reconfigurable platforms by demonstrating the performance improvements in various applications such as image processing, signal processing, pattern matching, cryptography, etc. Other research focused on improving reconfigurable logic technology from the hardware and software perspective without focusing on any particular application.

Very little work has been done on reconfigurable computing application specifically tuned to networking applications. We argue that networking is particularly well suited to reconfigurable platforms, as they can provide significant benefits. Our work demonstrates validity of this claim.

Most of the previously built FPGA based systems for telecommunications have been built for prototyping and experimenting; they have not been used as reconfigurable computing systems. One such system is ATTRACTOR[MSK+98], developed in NTT's Optical Network Systems Laboratories. Although the system utilizes reconfigurable hardware, its software can only support off-line upgrades and expansions. The designers of the ATTRACTOR system have implemented support for system expansions using the Web browser as the user interface. However, the system does not support the user-level programmability specific to active networks, and its primary purpose is to simplify system upgrades and expansions.

2.5 Design Space

Network embedded programmability requires both flexibility and high performance. As stated in Chapter 1, the goal of our research is to address the performance problems in the data plane, while keeping the system flexible. From the discussion presented in Section 2.3, run-time reconfigurable hardware offers both flexibility and hardware performance, but it has its limitations. Clearly, programmable logic can never be as complex and flexible as software and it can never reach the performance of a specifically designed hard-wired circuit (*e.g.*, ASIC)

Figure 2.1 is a visualization of a performance vs. flexibility design space. The figure is not shown to scale and should be used only to motivate the discussion in this section.

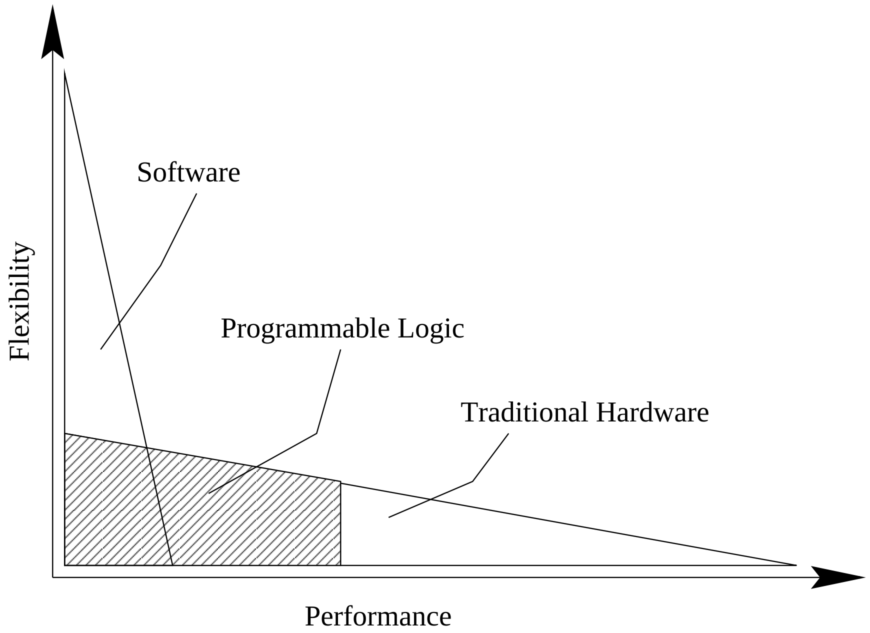

Figure 2.1: Design space covered by software and hardware systems

The unshaded portion of the lower triangle represents the subspace covered by traditional hardware. The rightmost end is covered by full-custom, hard-wired circuits optimized for a single function. They can achieve very high performance but offer no flexibility.

The shaded area represents the portion of the design space covered by programmable logic. It offers more flexibility than hard-wired circuits, at the expense of some performance. The fact that it covers an area not reached by the software systems (upper triangle) represents the opportunity for its use.

The goal of our research is to identify the functional elements in programmable networks whose performance can be improved with programmable logic without a major loss in flexibility. Since a large group of protocol processing functions are bit-oriented (*e.g.*, checksum calculation, FEC, encryption, etc.), the potential exists for speedups of orders of magnitude if these functions are implemented in hardware. A common property of this class of protocol processing functions is that they operate on packet contents and processing involves either payload transformations or calculations of some values based on the packet payload.

Many other protocol processing functions are packet-oriented (*e.g.*, ARQ, flow and congestion control, etc.) and thus amenable to software implementation. In this class of functions, packet arrival or its failure to arrive is the event that triggers the action and there is little or no processing on a packet payload. These functions are suitable for software implementation in the sense that little performance can be gained if they are delegated to hardware. This is in contrast to bit-oriented functions where a remarkable performance boost is observed on systems augmented by a special purpose hardware.

Many examples of packet oriented functions have been used in all-software active network prototypes to demonstrate the benefits of programmable network infrastructure. For example, a dynamic route discovery has been efficiently implemented using an active network based on PLAN[Hic99]. Network programmability is used in [LWG98] to implement retransmissions and NACK suppression for improved performance of reliable multicasting. The same paper also discusses the efficient implementation of on-line auctions using active networks. An implementation of a LAN bridge using active network is described in [ASNS97]. Examples reported in previous work on programmable networks show either acceptable or improved application performance. However, in all examples, the per-packet processing was light-weight (mostly examining the packet header or a few fields in the payload) without computationally intensive operations on packet content. In other words, example applications covered the portion of the design space which is suitable for software implementation and the measured performance overhead was sufficiently low to provide satisfactory results.

In our work we consider the cases which involve intensive bit oriented processing. In these cases, the processing overhead of the software is prohibitively high, but can be significantly reduced with the hardware assistance. Our work well captures this portion of the design space by allowing hardware implementation of the algorithms with near-software flexibility.

Project	Focus	Comment
X-kernel	Adaptation	Static protocol graph constructed off-line. Dynamic decisions about which parts to execute.
Protocol Boosters (kernel implementation)	Adaptation	Dynamic construction of protocol stack. High processing overhead for bit-oriented functions
XBind	Programmability	Programming limited to control plane only. No performance cost in data plane due to unchanged base network infrastructure. Discusses multimedia applications only.
ANTS	Programmability	Full programmability. Code downloading. High processing burden in network nodes could impact the performance.
Smart Packets	Programmability	Packet programs. Restricted programming language. Programming limited to network management plane to avoid performance impact.
Switchware	Programmability	Full programmability (packet programs and code downloading). Same performance issues as in ANTS. Good discussion of safety and security issues.
Hardware for Active Networks (non-P4)	Performance	Performance improved with hardware. Presented work mostly conceptual.
Protocol Boosters (P4 implementation)	Adaptation	Same functionality to original Protocol Boosters. Performance improved with hardware.
Protocol Boosters extension (P4 implementation)	Both	Adaptive infrastructure with added programmability. Hardware performance. Concerns about hardware safety and security.

Table 2.1: Comparison of Related Work

Chapter 3

The P4 Platform Architecture

The Programmable Protocol Processing Pipeline (P4) shown in Figure 3.1 is a run-time re-configurable FPGA board which we have built to serve as an experimental platform for our programmable networks research. It allows implementing protocol processing algorithms in hardware while maintaining software-like flexibility.

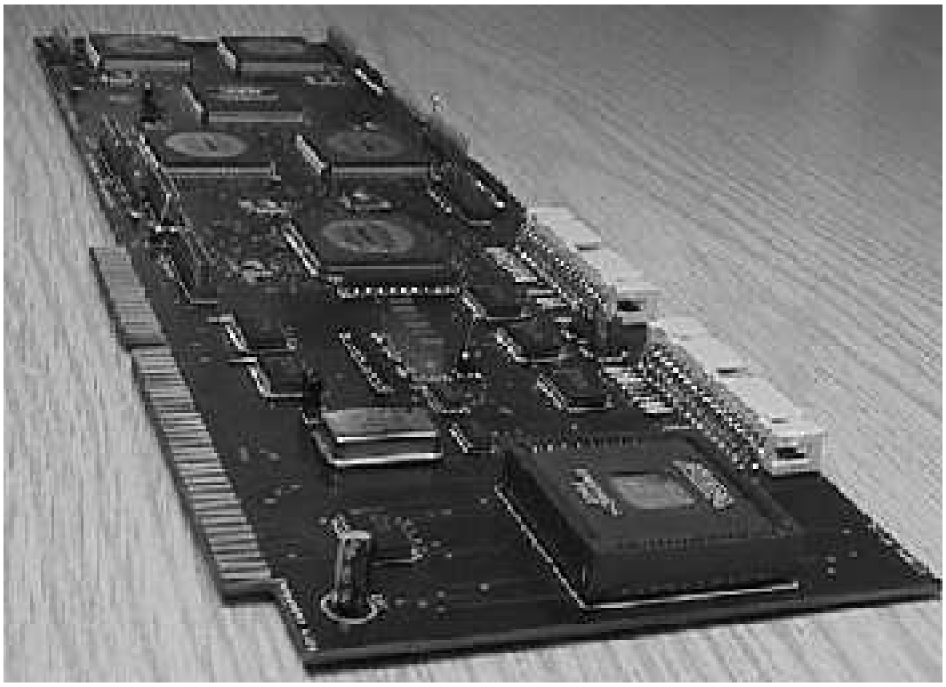

Figure 3.1: A photograph of the P4 board

During the course of our research we have implemented two versions of the P4 board.

This chapter describes the second, enhanced version. Details of the first version can be found in [HS97].

3.1 Basic Architecture

A block diagram of the P4 architecture is shown in Figure 3.2. The P4 platform is designed to operate on an OC-3 (155 Mb/s) ATM link[ATM94]. Its network interface (NIF) connects to an external SONET framer board[1], which implements the physical layer. The input to the P4 board is a stream of ATM cells. Cells are received by the receive section of the network interface, processed by the processing engine and sent back into the network. The processing engine may modify cell content, generate new cells or remove existing cells from the stream. The processing engine is the pool of the Altera Flex10K family[Alt98d] FPGA devices which act as processing elements (PE) and the switching array that provides interconnections.

The P4 prototype uses ATM cells as a convenient unit of processing. While the architecture is *not* ATM-specific, ATM simplifies the hardware implementation of processing algorithms and allows validation of performance in 100+ Mb/s operating regimes. The concept can be applied to any other protocol and is not limited to ATM, although some ATM features have been exploited.

The processing engine is programmed by configuring PEs with the desired FPGA configuration and switching them into the processing path. Processing elements typically form a pipeline "chain", linked via the switching array as shown in Figure 3.3. Each PE implements one function in the pipeline chain, but multiple protocol elements might be captured in each function. For example, if forward error correction was to be added into the protocol stack, an available FPGA device would be configured to implement the encoder on the transmitter side; on the receiver side, one of the PEs would be configured as the decoder.

Associated with each PE is a FIFO buffer. A PE reads the data from its FIFO buffer,

[1]The board used in our experiment is a non-commercial SONET board designed at Bellcore/Telcordia which in almost completely resembles the reference design for PMC-Sierra S/UNI interface chip[PMC96][PMC95]

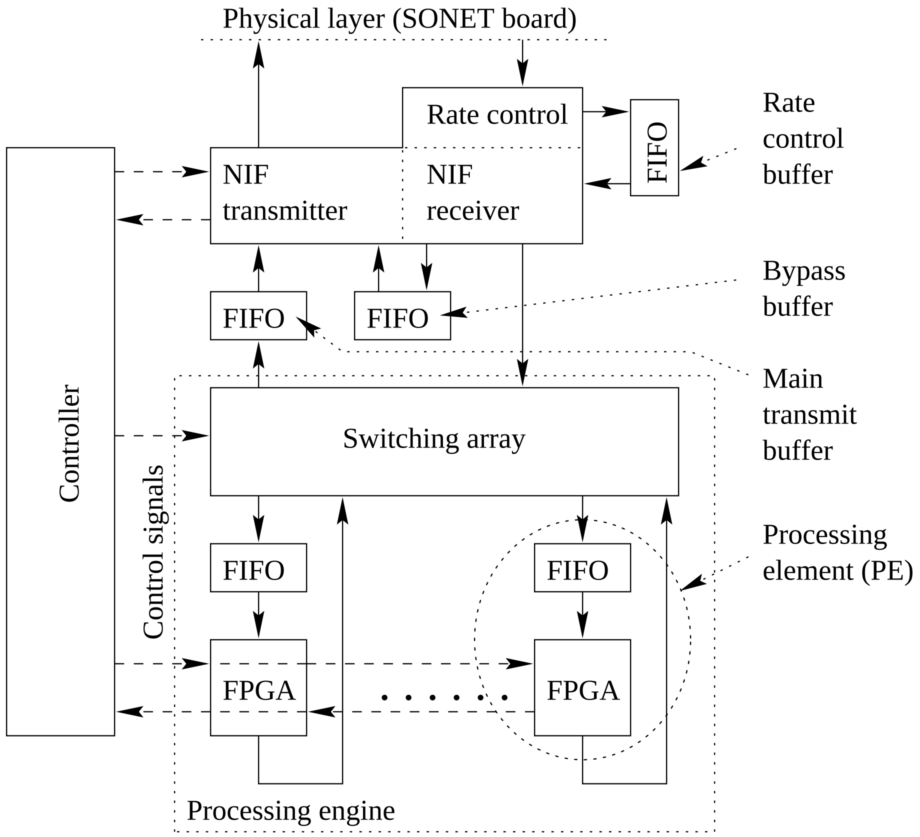

Figure 3.2: Block diagram of the P4 Architecture

performs its processing, and writes into the FIFO buffer associated with the next device in the chain. Connection to the next device is achieved via the switching array.

The first device in the chain is the receive section of the network interface. Its function is to demultiplex cells which belong to the virtual circuits processed by the board from other cell streams. It strips off most of the header and writes the payload into the FIFO buffer associated with the first processing element in the chain. Stream specific information (*i.e.*, the VPI and VCI) is forwarded to the transmit section of the network interface while the cell specific information (*i.e.*, cell type, cell loss priority and user-to-user indication field) is reduced to a one-byte preamble and passed to the processing engine before each cell. Cells not processed by the board are passed to the bypass FIFO buffer together with the original header.

After passing through the processing pipeline chain, payload data are written into

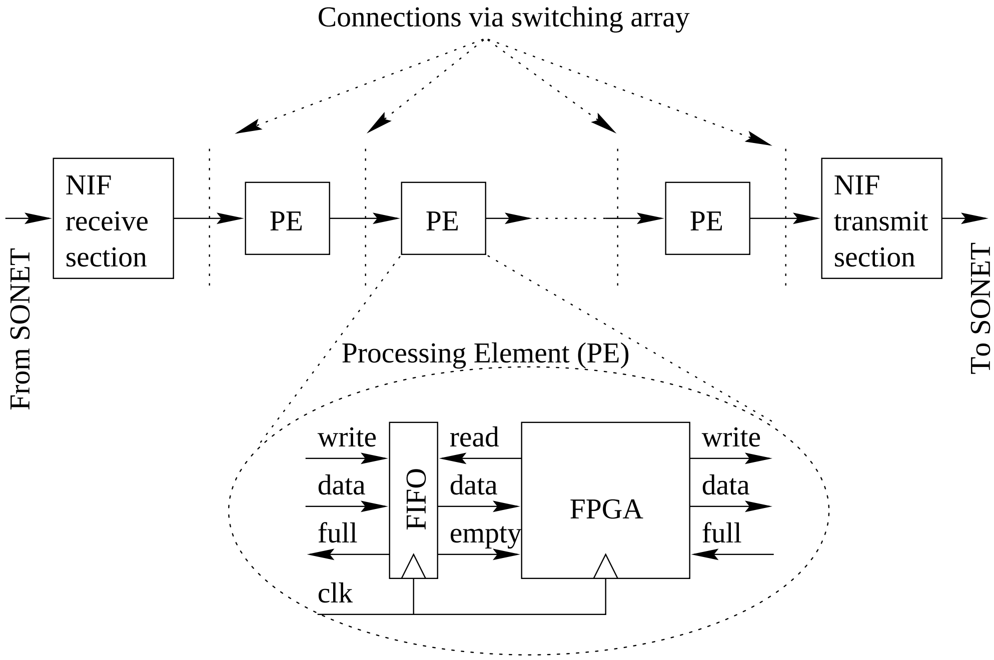

Figure 3.3: A pipeline chain, typical P4 topology

the FIFO buffer associated with the last device in the chain: the transmit section of the network interface. The payload data are mapped back into the stream of ATM cells and cell header is reconstructed using the information provided by the receive section and the preamble. The reconstructed stream is multiplexed with the cell stream from the bypass FIFO buffer.

The P4 board is controlled by a general purpose processor which we refer to as a "controller". The controller configures the FPGA devices and the switching array in the processing engine and implements the signalling protocol necessary to synchronize the operation of P4 boards distributed throughout the network.

In production the controller would be implemented as an embedded system or it would be a part of a switch. In our experiments we used a PC running Linux operating system (kernel version 2.0.34) and the board is seen by the controller as an ISA I/O device. Note that the data from the network (except for the low bandwidth signalling data discussed in Section 3.2.1) do not go through the controller bus. Instead, they are processed by the P4 board in hardware and sent back to the network independently of the traffic on the controller bus. The traffic on the bus includes configuration data, signalling and control signals, all with low bandwidth demands. This makes the slow, but robust ISA bus[SA95] an appropriate engineering choice.

3.2 Functional Elements

3.2.1 Network Interface (NIF): Receive Section

The receive section of the network interface (Figure 3.4) provides front end processing of the cell stream. It shares a specifically configured FPGA device with the transmitter section and it is downloaded on power-up. It consists of the input interface and the rate control unit. The input interface connects directly to the physical interface (SONET framer board) and synchronizes to the 19.44 MHz clock recovered from the network. Data from physical interface are received in 8-bit parallel form resulting in a 155.52 Mb/s line rate.

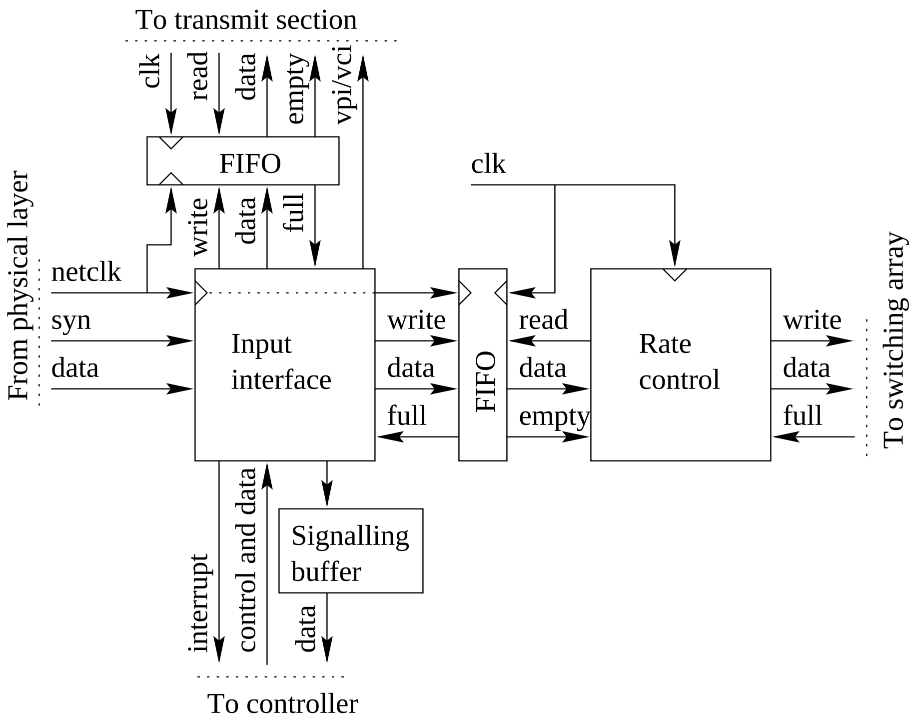

Figure 3.4: The network interface receiver section

During operation, the controller passes the VPI/VCI of the processed channel (used to make the demultiplexing decision) to the network interface. This configuration step is separate from downloading the FPGA on power-up. The same FPGA configuration is used for all virtual circuits and in addition, internal registers are programmed with the VPI/VCI. The current implementation does not support concurrent processing of multiple virtual circuits and the controller must set the new VPI/VCI when it wants to change the processed channel. Adding support for multiple virtual circuits would be a straightforward technical task, which would increase the complexity of the board without adding significant research value to the project. Even the potential performance impact of processing multiple virtual circuits, could be addressed by a skillful hardware designer without changing the basic concepts of the P4. Since we were able to perform all our experiments on a single virtual circuit, we decided to simplify the design by supporting only one virtual circuit. However, we recognize that in the real-world system, support for multiple virtual circuits would be an essential feature.

The input interface scans the received data stream for cells belonging to the processed virtual circuit and stores them in the FIFO buffer associated with the rate control unit. All other cells are stored in the bypass buffer together with the header fields.

Headers of cells on the processed stream are reduced to a single byte preamble which carries the payload type, cell loss priority and user-to-user indication bit. The preamble and the payload data are stored into the rate control FIFO buffer, while the virtual circuit information is passed directly to the transmitter section via a special data path, bypassing the PEs. If the rate control FIFO is in an almost-full state (unable to accommodate an entire additional cell) the incoming cell is dropped to prevent the buffer overflow.

There are two reasons for stripping off the header fields:

1. Having processing elements deal with only the payload reduces the complexity, saving space in the FPGA device and enabling us to put more *processing* functionality in a PE.

2. Some processing algorithms may change the size of the actual data. For example, the output of the data compression will make the data stream shorter, resulting in fewer cells. For this and similar cases it is clearly simpler to remove the header before the processing and reconstruct cells at the output.

The rate control unit reads the data from the FIFO synchronized to the internal 19.44

MHz clock. It provides two major functions:

1. Cells stored at the clock rate synchronized to the network are read out at the clock rate synchronized to the board. The rate control FIFO buffer acts as an elastic buffer compensating for the difference between the network and board clock rates. Although the network and board clock operate at the same nominal rate, the lack of their synchronization would eventually result in either data starvation or loss if the rate control buffer were omitted.

2. The rate control buffer implements a leaky bucket algorithm[Tur86] with parameters (token arrival rate and token capacity) programmable by the controller. This feature may be used to shape the load inside the processing engine if necessary. A typical use of load shaping would be in cases when the processing engine creates additional payload data (*e.g.*, the FEC encoder discussed in Chapter 4). Since the processing engine operates on a byte stream of payload data, an overload would result in a loss of an arbitrary number of bytes which would further result in a loss of alignment after the cell stream is reconstructed. Using the leaky bucket in front of the processing engine could shape the input traffic so that the entire input cell is dropped before the buffers in the processing engine become full. This way, a unit of data loss would be an entire cell (whose size is known) and the cell alignment would not be lost.

At the output of the rate control unit, the cell consists of a one byte preamble and 48 bytes of payload as shown in the Figure 3.5. Every 49^{th} byte of data is a preamble and only the lowest three bits are used. The upper five bits are always set to binary value 10100 which can be used to check for cell integrity which we discuss in Section 3.2.2.

Since there are four bytes less than in the original cell, the processing engine can afford four idle cycles after it has processed the cell and before it must start processing the next cell. In addition, path and section overheads at the SONET layer allow even more additional cycles if the need arises. Our experience has shown that the implementation of protocol processing algorithms is significantly simplified if these extra cycles are allowed. In most cases no more than two extra cycles were needed and processing engine was never a bottleneck in the data path.

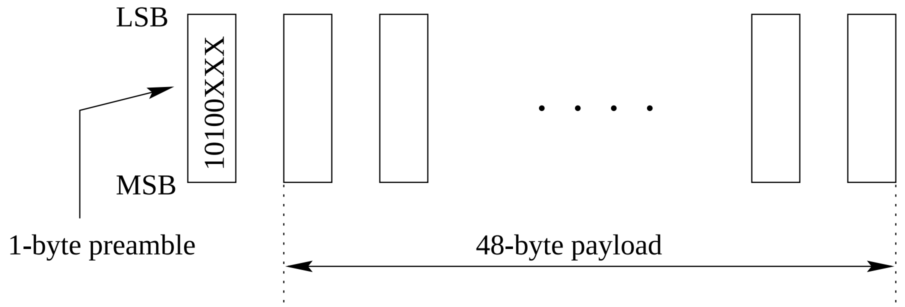

Figure 3.5: Cell format in the processing engine

A form of signalling protocol is necessary to synchronize the operation of multiple P4 boards distributed over the network. We use cells with the payload type field set to 111 binary (reserved payload type)[ATM94] to provide a signalling channel within the processed virtual circuit. The input interface filters out cells that match the programmed virtual circuit and have 111 in the payload type field. Signalling cells are not passed to the processing engine. Instead, they are stored into the signalling buffer and an interrupt is generated to the controller. The controller responds to the interrupt by reading a cell from the signalling buffer. Since the bandwidth demands of the signalling traffic are very low, using interrupt driven I/O is an appropriate solution. The received signalling cell is passed to the signalling protocol which is implemented in software and executes on the controller. We will discuss the signalling protocol in Section 4.4

3.2.2 Network Interface (NIF): Transmit Section

The function of the transmit section complements that of the receiver. The device reads from the FIFO buffer associated with it, reconstructing the processed cell stream using the header information fields provided by the header forwarding data path. the reconstructed cell stream is multiplexed with cells from the bypass buffer (data streams not processed by the P4) using rotating priorities. If both buffers have data available, the buffer which has been read last has lower priority. The output connects to the SONET board and is clocked out in 8-bit parallel format at 19.44 MHz (*i.e.*, 155.52 Mb/s line rate). If no data are available, SONET board will insert an idle cell to maintain cell synchronization. The transmit section operation is illustrated in Figure 3.6.

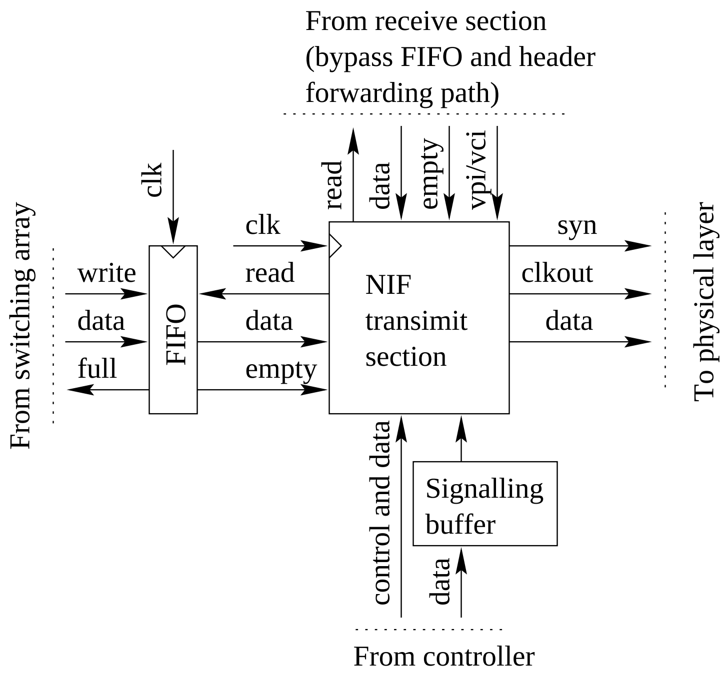

Figure 3.6: The network interface transmit section

Data read from the main FIFO buffer are constantly checked for cell integrity using the known value of the upper five bits of the preamble. As long as the transmitter is reading the binary value 10100XXX before each cell it assumes that the cell synchronization is maintained. Failure to correctly read the preamble will result in the alarm state. The transmitter will attempt to recover by searching for five consecutive candidates for the preamble placed 49 bytes apart. During the recovery phase, data will be read from main buffer, but main cell stream will not be reconstructed. Cells from the bypass buffer will be normally read and transmitted as their integrity has not been violated.

In addition, there is a special control input pin which can be used to delay the transmitter in generating the cell even if the main buffer is not empty. If a delay has been requested, the transmitter will act as if the buffer is empty and either insert the cell from the bypass FIFO or let the SONET board insert an idle cell. The transmitter can be delayed by any processing element in the chain. For example, if one of the processing elements performs data compression, it may request a delay in cell generation if it produced the output shorter than 48 bytes before new data appeared. In this case there are not enough bytes to complete the cell generation and delaying the cell for a limited amount of time might avoid padding the payload with idle data and improve the bandwidth utilization.

Support for signalling in the transmit section is implemented with the signalling buffer which can be written by the controller. Writing to the signalling buffer will initiate signalling cell generation with binary 111 in the payload type field. The signalling cell has the highest priority and it will be generated ahead of other cells from the bypass or main FIFO buffer. However, it will not preempt the cell in progress.

3.2.3 Switching Array

The purpose of the *switching array* is to improve flexibility of the P4 architecture because in general functions in the PEs are not commutative. If a new processing element must be activated, it can be downloaded into any available FPGA device. The switching array will allow placing of the processing element at the appropriate point in the pipeline chain regardless of the actual FPGA device used; it serves to virtualize pipeline ordering.

One example that illustrates the need for the switching array is the combination of FEC encoding and data compression. Suppose that the data compression is to be added to the processing engine with the FEC encoder already in the processing path. In this combination of processing elements, compression must be performed before the encoding if the FEC is to be useful. If there were no switching array, and the encoder resided in the first device in the chain, the controller would have to clear the first FPGA device, load it with the new configuration (data compression) and load the next device in the chain with its configuration (encoder). In this case, the time to add the data compression to the protocol stack equals twice the FPGA configuration time. With the switching array, the compression module can be downloaded into any available FPGA device and the order of processing can be rapidly interchanged by reprogramming the switching array. This reduces the time to add the function to the protocol stack to the time to download a single FPGA device.

The switching array can also exclude the unused processing elements from the processing chain. The processing element may be programmed in advance should the controller anticipate that a function will be needed soon and kept preloaded until needed. Using the switching array, the controller can quickly insert the processing element into the processing path, as setting up the switching array is several orders of magnitude faster than downloading the FPGA device.

3.2.4 Processing Element (PE)

The processing element (PE) is the generic part of the architecture. It is a combination of a FIFO buffer and the FPGA device. The output of the FPGA device is the set of control and data signals compatible with the FIFO buffer input. Processing consists of reading from the FIFO buffer, processing the data and writing it into the FIFO buffer associated with the next PE in the chain.

In the minimal case (no PEs active), network interface reads the data from the SONET board and writes into the transmit buffer. The boundary between PEs is the FIFO interface. Since this is well defined for the given FIFO it is simple to insert and remove PEs in the P4's chain.

3.2.5 Controller

From the controller's perspective, processing elements are I/O devices. The controller has a pool of the FPGA configurations with which it can implement a processing algorithm. Depending on the policy and received signalling messages, it chooses the appropriate set of configurations and downloads the FPGA devices on demand. Downloading is managed by the controller, and the FPGA is in the passive configuration mode[Alt98b]. Using the switching array, the controller creates the chain with processing elements ordered appropriately. We used a PC running the Linux operating system as the controller due to its convenient debugging and experimental environment. The FPGA configurations are in files stored on a disk. If the P4 were used in a real network rather than as an experimental infrastructure, configurations might be stored on some faster storage device to reduce the download time and improve the system "agility".

3.2.6 P4 Role and Placement in a Network

The P4 board is intended to work as a front end device at the access point to the P4 augmented portion of the network cloud as shown in Figure 3.7. Two P4 boards are necessary to process bidirectional traffic (one board in each direction). Although the prototype is designed to operate on a stream of ATM cells, the same concept can be applied to any type of packet stream.

The P4 can work as a stand alone unit with its own controller or as part of a switch. In the latter case, the P4 can be an integral part of the switch and thus managed by the controller in the switch. A single controller might be responsible for multiple P4 boards.

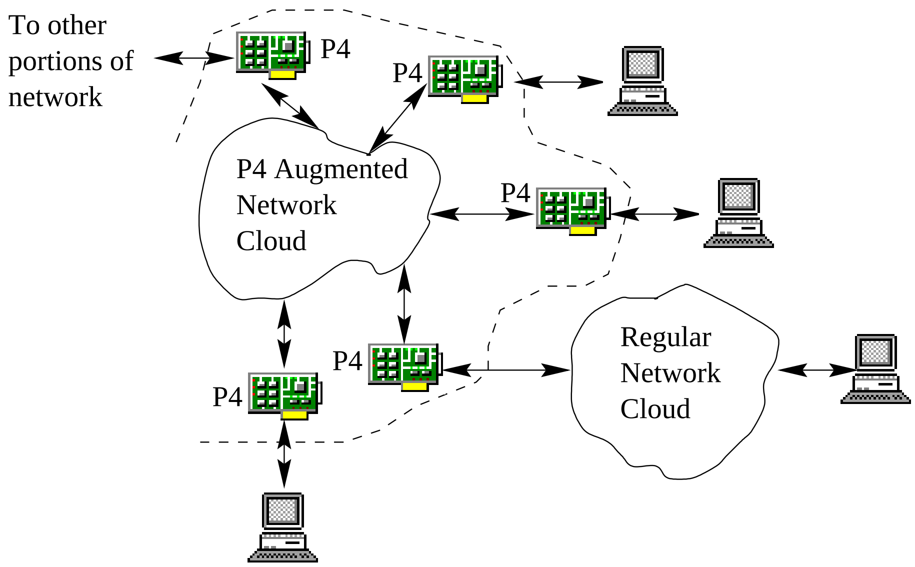

Figure 3.7: P4 as the part of the network

In Figure 3.7, the P4 augmented portion of the network has a P4 board associated with all access points. If a user connected to the P4 enhanced access point communicates with another user connected to a P4 enhanced access point, their communication would be as if portions of the host protocol were implemented in hardware, provided that there is a route entirely through the P4 augmented portions of the network. An important property of the P4 as used in this mode is that if the route has to pass through the non-P4 portion of the network, the P4 board at the boundary will act as the conversion unit by converting the outgoing traffic to the original protocol and applying its processing to incoming traffic.

Chapter 4

Case Study: FEC Booster

In this chapter we study the application of the P4 in constructing an adaptive protocol using programmable hardware. Preliminary results of this Chapter have been previously reported in part in [HSM98] and [Had98]. This chapter extends and updates the results from our earlier work, and provides a complete study of an example adaptive protocol.

We follow the Protocol Boosters[FMS+98] model as it provides a convenient policy/mechanism separation. Mechanisms are well defined functions transparent to the application and the outcome of the policy decision is binary: insert or remove the mechanism.

We use TCP/IP as a base protocol and apply the Protocol Boosters design methodology to adapt the protocol in face of network errors. We do this by dynamically inserting and removing FEC on an as-needed basis.

We use the FEC as an example of a simple protocol processing function whose performance can be significantly improved by implementing it in hardware and which clearly demonstrates the benefit of protocol adaptation. Running TCP without FEC reduces its operating range as it does not deliver the acceptable performance for high bit error rates. However, for low bit error rates, FEC represents an unnecessary overhead.

In our system, hardware implementation of FEC is inserted only when necessary. If the FEC is not needed, the resources on P4 (*i.e.*, FPGA device) can be used for other functions. Assuming that there exists another bit-intensive function currently in the protocol stack the same P4 hardware can be reused, after the FEC is removed. Without P4, the use of FEC would be either limited to software implementation with the associated performance penalties, or a hard-wired circuit which is inflexible and represents an idle resource when the FEC is not used.

We begin by describing our implementation of FEC encoder and decoder on P4. The code used in our experiments is a simple textbook example[PS94]. Since our focus is on demonstrating the benefits of adaptation and means for achieving it, we are not concerned about the particular error model, except noting that this is an important issue in real-world systems.

4.1 Mechanism: Encoder and Decoder

Data are encoded with the rate $R = 1/2$, constraint length 3, convolutional encoder[1] with generator polynomials $g_1 = x^2 + 1$ and $g_2 = x^2 + x + 1$. Bits of each data octet are grouped in four chunks of two bits and encoded independently using four parallel encoders. Each encoder accepts two bits from the current octet and produces four output bits in a single clock cycle. Four parallel encoders thus produce 16 bits of data, which are mapped into two consecutive octets as shown in Figure 4.1, and clocked out at 19.44 MHz, resulting in the output bit rate of 155.52 Mb/s. To avoid dropping cells of the input stream, the average load must be lower than 77.76 Mb/s.

On the decoder side, four parallel Viterbi decoders operate independently on the groups of two bits producing one bit of the original data stream per clock cycle. One input octet produces the four-bit decoded word. Outputs of two consecutive input octets are combined into one decoded octet as shown in Figure 4.2, complementing the encoding operation.

At the cell level the encoder creates two cells per input cell and the redundancy is scrambled into the original data. The first output cell is a result of encoding the first 24 bytes of the original cell and the second cell is the result of encoding the last 24 bytes. The decoder reconstructs the original cell out of two encoded cells.

[1]For background on coding theory, see [LC83].

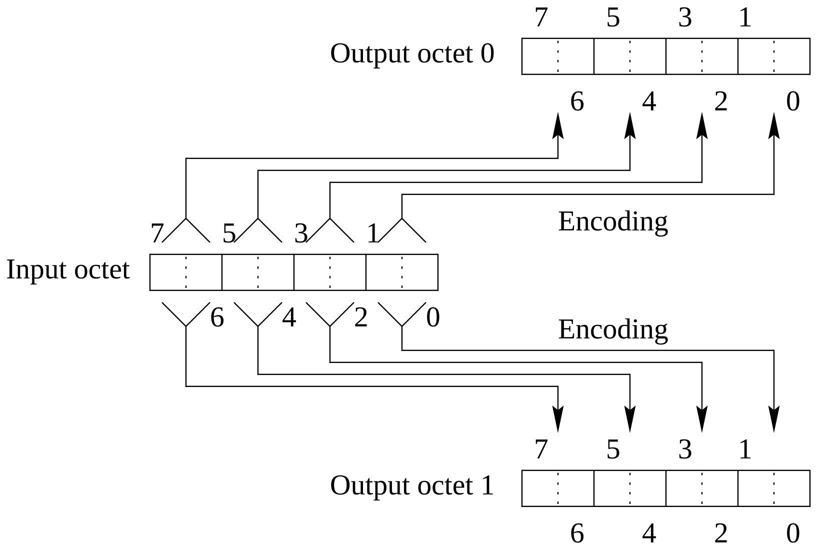

Figure 4.1: Mapping the original data stream to an encoded data stream

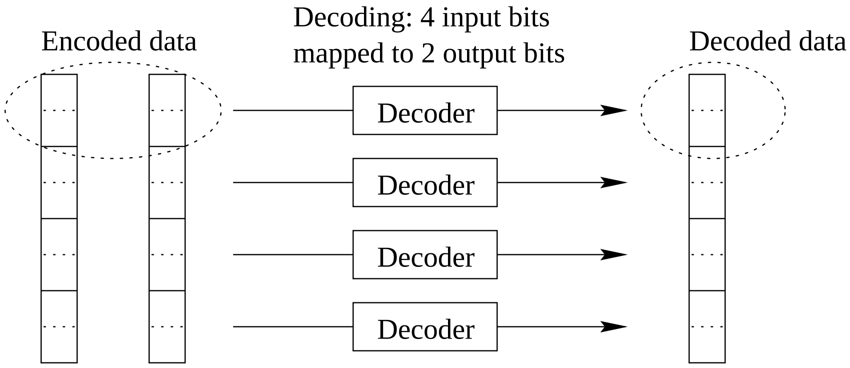

Figure 4.2: Mapping the decoded data stream to original data stream

4.1.1 Robustness

An important issue in protocol design is robustness. Although it protects user data from bit errors, convolutional encoding may increase the sensitivity to other impairments such as cell losses and cell misinsertions. In general, the output of the Viterbi decoder depends on the history of its inputs. If a cell is lost, missing data may cause unpredictable behavior, and the error can propagate far into the future. To improve robustness in such cases and prevent error propagation, the encoder resets its state after encoding 24 bytes of input data (half the ATM cell) and the decoder resets its state after decoding each cell.

Both cells generated by the encoder have the same value for the user indication bit in the ATM header which is a copy of the user indication bit in the original cell. If the encoded cell within the AAL-5 protocol data unit (PDU) is lost, there will be a mismatch in the user indication bit at the end of the AAL-5 PDU. Prior to decoding, data are passed through the front end processing unit which checks for matching user indication bits. Only pairs of cells that match are passed for further decoding. If a mismatch is found, an all-zero cell with the appropriate user indication bit will be inserted as shown in Figure 4.3. This will, of course, result in a series of bit errors after decoding, but will prevent any error propagation that might otherwise result. Only the AAL-5 PDU whose cell has been lost will be affected. It can be easily verified that the front end processing unit will also successfully isolate bad AAL-5 PDUs in the case of cell misinsertions, thus avoiding error propagation.

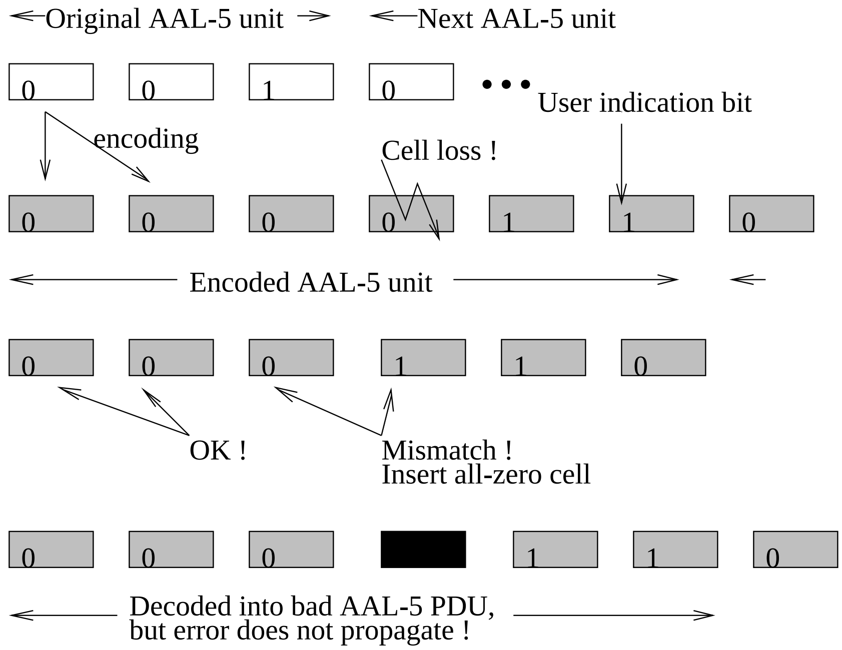

Figure 4.3: Protection from error propagation due to cell loss

4.2 Static Measurements: Mechanism Evaluation

Our first experiment evaluates the effect of the P4 implementation of FEC on the TCP throughput. With a tunable bit error rate induced on the link, we measured the TCP throughput seen by the receiver with and without the FEC booster. The model of TCP performance in face of packet losses derived by Padhye[PFTK98] predicts the exponential drop in throughput as packet loss rate increases. This is due to TCP's congestion control mechanism which assumes that every loss is due to the network congestion and backs off. In a noisy environment, the major source of packet loss is bit error rather than congestion which makes the TCP congestion control mechanism inappropriate.

Forward Error Correction (FEC) provides a low level mechanism for reducing the the number of packet losses due to bit errors. Despite the increased bandwidth demands there exists a bit error rate above which TCP throughput is improved by the addition of FEC. We experimentally verify this claim using the P4 implementation of the convolutional encoder and the Viterbi decoder.

4.2.1 Test Setup

The experimental setup is shown in Figure 4.4. The workstation is a 300 MHz Pentium II processor with 64 MB of 60 ns DRAM, 16 KB of L1 cache, 256 KB of L2 cache and a 33 MHz PCI bus. It operates Linux 2.0.29 patched with the support for ATM[ATM] and uses a Fore Systems PCA200E ATM adaptor[For]. The P4 is controlled by 166 MHz Pentium running Linux 2.0.34 and the P4 control software, which we use as a substitute for an embedded controller.

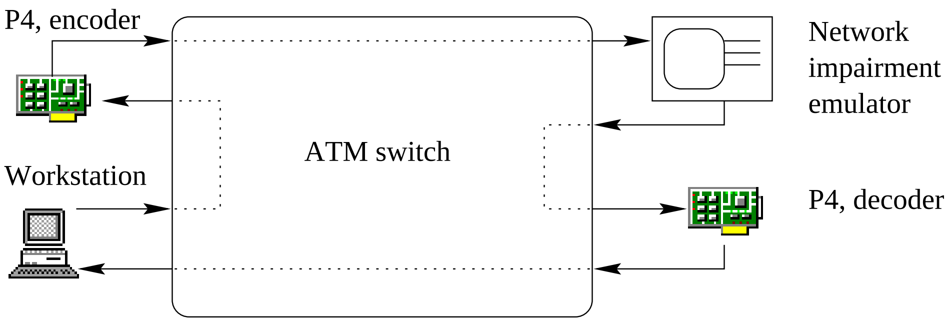

Figure 4.4: Experimental setup used in testing the FEC booster

The logical ATM link operates at the SONET OC-3c rate of 155 Mb/s. In reality, the logical link consists of several subsegments, starting from the transmit port of the adaptor:

1. The link to the Fore ASX-200 ATM switch

2. The link to "encoder" P4 and the link back to the switch

3. The link to the Network Impairment Emulator[Dim97] (it is a VXI card in an HP 75000 Broadband Network Analyzer and is used to emulate a noisy line by inserting bit errors) and the link back to the switch

4. The link to "decoder" P4 and the link back to the switch

5. The link from the switch to the workstation adaptor

Throughput testing is done with `ttcp`. For convenience, we used a single test machine with source and sink running as two separate processes. Since we were interested in testing the relative impact of the P4 on TCP throughput, not the impact of the workstation, the effects of using a single workstation (*e.g.*, process synchronization) are of a secondary importance and this setup can deliver useful results.

Cells transmitted by the workstation are encoded using the first P4 in the test setup. At the output of the first P4, the utilized bandwidth is twice the bandwidth generated by the workstation due to the additional cells. To prevent cell drops in the operating P4, the device driver in the workstation must be rate controlled.

We implemented the rate control by modifying the device driver to force a tunable idle period between the transmission of two consecutive AAL-5 packets. We use the following procedure to tune the rate control mechanism:

1. The network impairment emulator is set to error-free mode.

2. The P4 boards are set to pass the traffic without any processing (idle P4) and rate control is turned off at the workstation (*i.e.*, the workstation generates the traffic with the maximum bandwidth it can achieve).

3. The workstation runs a long `ttcp` test session so that the traffic load is constant for an extended period of time.

4. The link bandwidth utilization after the first P4 is recorded using the HP75000 Broadband Network Analyzer.

5. The P4 boards are set to run FEC encoder and decoder with rate control turned on at the workstation and measurements are repeated.

6. The rate control parameter (i.e., length of the idle period) is varied, and the test repeated until the bandwidth used on the link after the encoder becomes equal to the bandwidth used without the FEC in place.

This method ensures that the combined traffic load generated by the workstation and P4 running FEC does not exceed the traffic load generated by the workstation alone. It also ensures a fair comparison between measured throughput with and without FEC as the workstation alone may not be able to fill the OC-3 pipe. In such cases, running the test without the rate control would allow some "free" bandwidth for the FEC. Rate control chokes the workstation when the FEC is inserted into the protocol stack, so that the load generated after the encoder does not exceed the bandwidth a workstation alone is capable of using.

Encoded cells are passed through a noisy link, emulated by inserting bit errors with the Network Impairment Emulator[Dim97]. We vary the bit error rate and measure the TCP throughput seen by the receiving process on the workstation with and without the FEC in place. The second P4 board decodes the cells and corrects any correctable bit errors.

4.2.2 Results

We first verified if the P4 hardware itself induces any overhead. We ran the `ttcp` test with both the TCP and UDP protocols and varied the packet size. We repeated the experiment in the system without P4 boards, and with P4 boards present but doing no processing. Results for the TCP and UDP tests are plotted in Figure 4.5 and Figure 4.6 respectively. The overhead of the P4 hardware is minimal, the expected result since the P4 is a fully pipelined architecture capable of preserving throughput at an OC-3 line rate.

Next, we ran the `ttcp` throughput tests for four cases:

1. With P4 boards doing no processing;

2. With P4 boards doing no processing, rate control on;

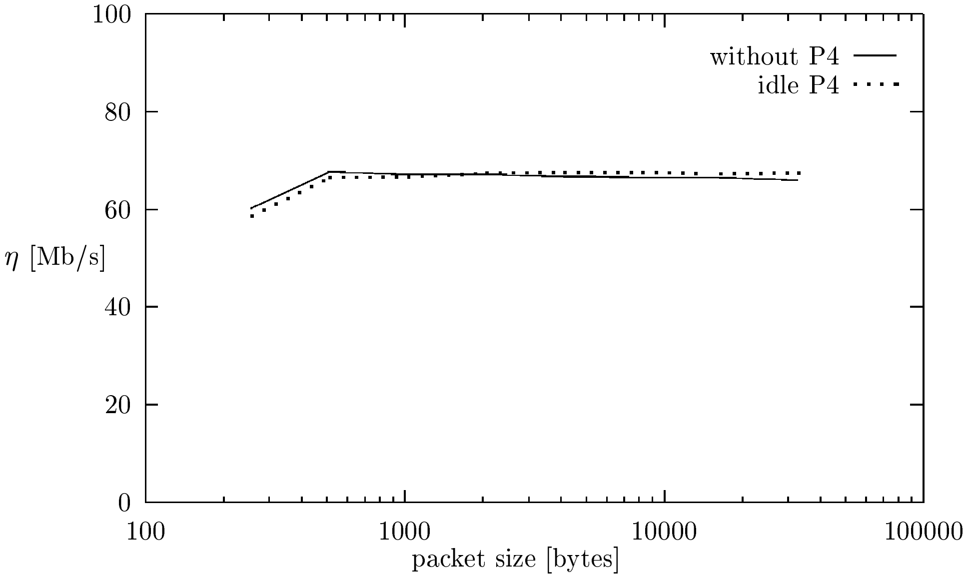

Figure 4.5: Comparison of measured TCP throughput without P4 and with idle P4

3. With P4 boards configured as FEC encoder and decoder;

4. With the FEC implemented in software.

We varied the bit error rate (BER) from 10^{-12} to 10^{-4} with an exponentially distributed time between two consecutive errors. The results are shown in Figure 4.7 and Table 4.1 Results from the first case provide a baseline measurement. We observe an enormous dropoff in TCP/IP performance in the face of bit errors (the throughput is on a logarithmic scale in Figure 4.7). This is due to TCP/IP's strategy, which is to assume that the packet loss was a result of congestion rather than noise. The result is that the TCP/IP congestion window is rapidly reduced to the point where the protocol becomes "stop-and-wait", with the consequences shown in Figure 4.7. The second case shows the effect of rate control, namely that the throughput starts off considerably lower but drops off as rapidly as the first case in the face of error. The reason this test was performed was to separate the costs of rate control from the costs associated with the FEC processing.

In the third experiment, we measured TCP throughput with the FEC in place and rate controlled, as before. Very little overhead is due to the FEC processing and the dominant cost is in the rate control. This clearly demonstrates the benefit of using hardware, which has reduced the processing cost to a small level, leaving only the irreducible cost of

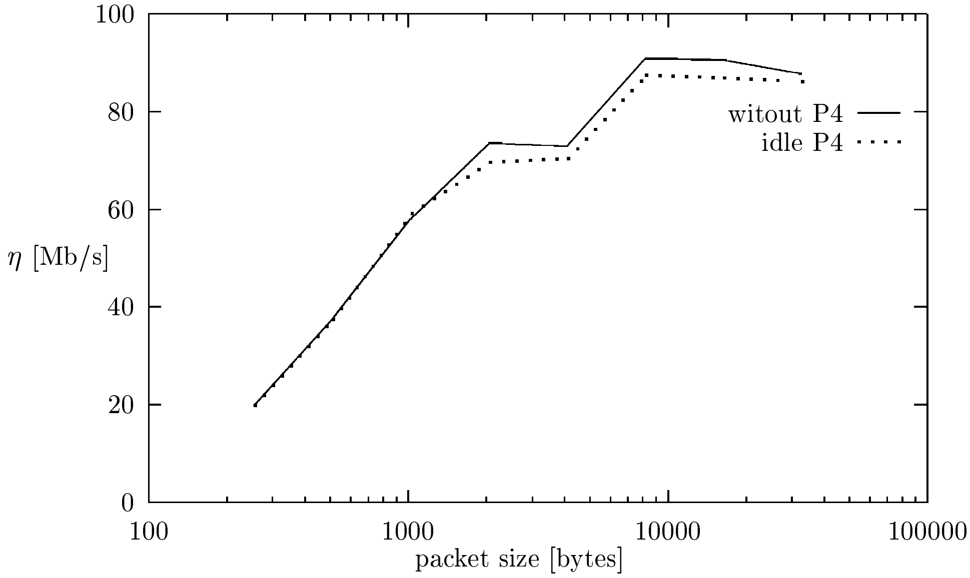

Figure 4.6: Comparison of measured UDP throughput without P4 and with idle P4

additional bandwidth necessary to transmit the redundancy generated by the FEC.

In the high BER region, the TCP protocol stack benefits from FEC in reducing the number of retransmissions and keeping the value of the TCP window size larger. Without FEC, TCP completely stalls at a BER below 10^{-4}, while it is still able to operate with FEC in place.

The last experiment demonstrates the performance implication of implementing the same code in software. We repeated the experiment without the P4 boards and FEC implemented in software. The FEC code was the same as the one implemented in hardware with some minor unavoidable differences:

1. The encoding was done over the entire AAL-5 packet rather than the cell. If we wanted to encode the data at the cell level, we would not be able to use the hardware segmentation and reassembly (SAR) on the network card which would unfairly reduce the system performance since all other test cases were able to take the advantage of hardware SAR.

2. If the packet incurs an error, the AAL-5 CRC will fail at the receiving network interface card and the software implemented decoder will not see the packet. Hence, we had to use the P4 board following the Network Impairment Emulator to recalculate

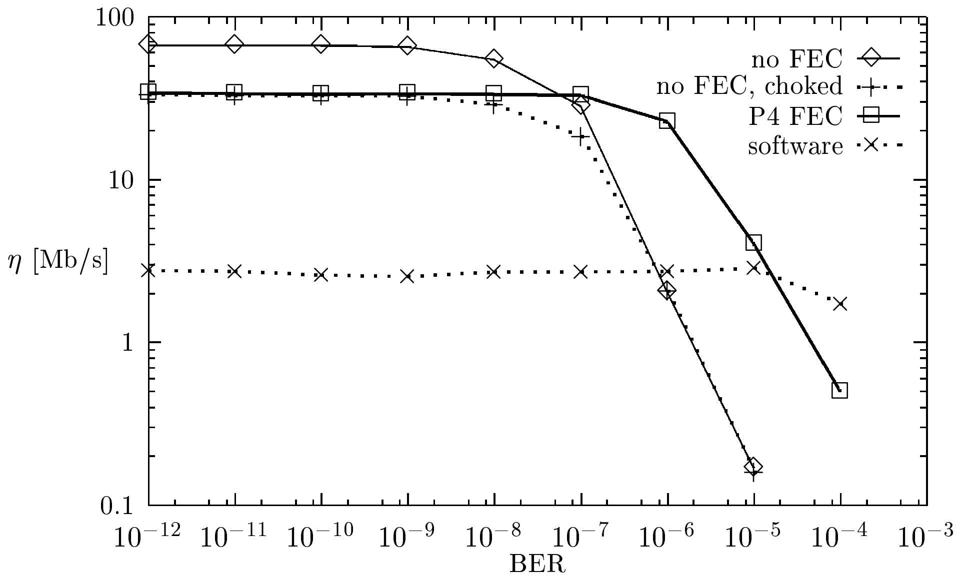

Figure 4.7: Mean value of measured throughput

the CRC and ensure that the packet is not dropped by the card.

In addition we did not reset the encoder every 24 bytes but at the end of the AAL-5 PDU instead. Since the decoding is done *after* the AAL-5 packet has been reassembled, there is no potential error propagation in the case of a cell loss. This implementation difference gives some advantage to the software because when resetting the decoder, as all remaining bits in the decoder buffer must be decoded which weakens the ability of the decoder to correct bit errors[2]. In the software implementation, decoder resetting is done once per AAL-5 packet, while in the hardware implementation, resetting is done twice per cell.

The rate control was not necessary in this case because the FEC processing already provides sufficient choking at the workstation. The benefit of FEC can be observed only for extremely high bit error rates when the plain TCP completely stalls. However, the processing cost of encoding and decoding is so high that the region in which the FEC is helpful decreases dramatically. This illustrates the advantage of hardware implementation.

Figure 4.7 shows the logarithmic plot of the mean value of measured throughput as a function of BER and Table 4.1 shows 90% confidence intervals for measured throughput.

[2]Detailed discussion is in Appendix B

BER	no FEC	no FEC, choked	with FEC	software FEC
0	$[66.74, 67.09]$	$[32.87, 33.21]$	$[33.89, 34.05]$	$[2.67, 2.86]$
10^{-12}	$[66.61, 66.88]$	$[32.96, 33.18]$	$[33.89, 34.03]$	$[2.68, 2.83]$
10^{-11}	$[66.58, 67.01]$	$[32.60, 33.27]$	$[33.86, 33.94]$	$[2.63, 2.84]$
10^{-10}	$[66.71, 67.01]$	$[32.56, 33.25]$	$[33.80, 34.97]$	$[2.54, 2.64]$
10^{-9}	$[64.58, 67.01]$	$[32.25, 33.07]$	$[33.44, 34.09]$	$[2.54, 2.55]$
10^{-8}	$[51.61, 57.92]$	$[27.27, 30.67]$	$[33.13, 34.00]$	$[2.62, 2.81]$
10^{-7}	$[21.14, 35.14]$	$[16.94, 20.06]$	$[32.31, 33.43]$	$[2.60, 2.81]$
10^{-6}	$[1.97, 2.13]$	$[1.98, 2.14]$	$[21.44, 24.00]$	$[2.68, 2.79]$
10^{-5}	$[0.17, 0.18]$	$[0.15, 0.16]$	$[3.80, 4.24]$	$[2.39, 3.36]$
10^{-4}	N/A	N/A	$[0.47, 0.53]$	$[1.68, 1.75]$

Table 4.1: Confidence intervals (90%) for measured throughput [Mb/s]

Log plots are used since the BERs of interest cover many orders of magnitude. The line marked with $\diamond$ shows the throughput of the unmodified TCP, while the line marked with $+$ shows the throughput without FEC but with rate control turned on. An enormous dropoff in throughput for high bit error rates is due to TCP's congestion control mechanism, as discussed earlier. The third line marked with $\square$ demonstrates the benefit of adding FEC as the throughput improves for high bit error rates despite the rate control and bandwidth demands of the FEC encoding. We also observe that the major cost is in rate control, not the processing. Finally, the line marked with $\times$ demonstrates the benefits of hardware implementation as the processing overhead introduced by the software makes the FEC practically useless, except for very high bit errors where the advantage of reseting the encoder less frequently becomes dominant. For BERs greater than $5 \cdot 10^{-8}$, TCP gains from FEC. Given our earlier explanation of TCP's response to packet loss, it should be clear that the FEC, in reducing the impact of noise, reduces the probability of the incorrect assumption of congestion. Thus, the performance is improved.

The graph in Figure 4.7 illustrates an opportunity for an adaptive protocol. In particular, the intersecting curves at a BER of about $5 \cdot 10^{-8}$ suggest that FEC be employed only when the BER exceeds $5 \cdot 10^{-8}$. In other words, FEC represents an unnecessary overhead when the bit error rate is low, while it improves the performance for BER above $5 \cdot 10^{-8}$. Thus, if the bit error rate varies, a policy module may constantly monitor the conditions on the link and switch on the FEC when needed.

Figure 4.8 compares the line followed by an ideal adaptive protocol with the throughput of unmodified TCP. The ideal throughput graph results from following the throughput curve with no FEC for low bit error rates and switching to the curve with FEC for bit error rates above $5 \cdot 10^{-8}$.

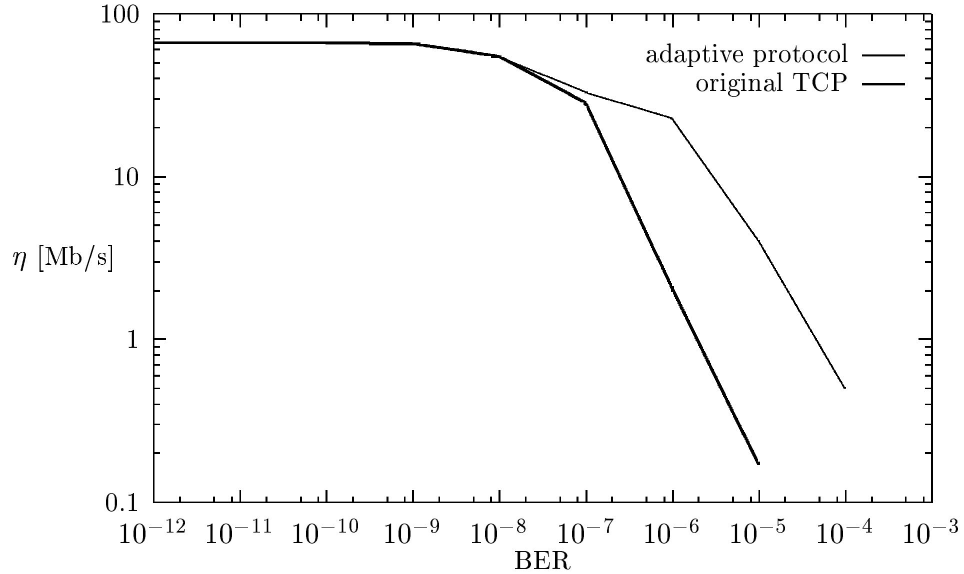

Figure 4.8: Throughput of an adaptive protocol vs. TCP throughput

In the Protocol Boosters framework, FEC processing is a *mechanism*, under control of the associated *policy*. The ideal throughput would result if the policy module never made an inaccurate decision and if the mechanism insertion and removal were infinitely fast. Obviously, an ideal throughput is not achievable in practice, but as we will show in Section 4.5, significant performance improvements can be achieved with a real-world policy module.

4.3 Policy

Static experiments identify only the *opportunity* for adaptation but they do not show how the opportunity can be exploited. The Protocol Boosters model calls for associating a policy module with the mechanism module. Based on measurements on the link, the policy module makes a binary decision: insert or remove the FEC mechanism. In this section, we describe our implementation of the policy module associated with the FEC mechanism.

The FPGA implemented bit error monitor (BER monitor) measures the number of corrupted packets inside a time window W. For each packet, a moving average sum is calculated (a recursive version is used in the implementation):

$$s[k] = \sum_{i=0}^{W-1} r[k-i] = s[k-1] + r[k] - r[k-W] \tag{4.1}$$

where $s[k]$ is the output for the k-th packet, W is the size of a moving average window, and $r[k]$ indicates if the packet is in error ($r[k] = 1$ for packet in error and 0 otherwise). Clearly, if we could identify bad packets, the output of a BER monitor would be proportional to the packet error rate.

Determining if the packet has incurred an error is specific to the underlying network infrastructure. To measure a bit error rate on a link without FEC, we perform the CRC check of the AAL-5 unit in which the packet is encapsulated. If the network is using early packet discard (EPD)[RF94], AAL-5 CRC is a good indicator of bit errors since packets are entirely dropped in the case of network congestion and almost all CRC errors will be due to bit errors rather than cell losses. Without EPD, CRC error can occur in the case of a single cell loss due to network congestion. In this case the BER monitor would make an assumption contrary to the one made by TCP (TCP assumes that all errors are due to congestion, while BER monitor assumes that all errors are due to bit errors).

When the FEC is on, using CRC check after decoding, is an inaccurate criterion. Since many bit errors are now corrected, passing the CRC check still does not necessarily imply that the bit error rate is sufficiently low to remove the FEC. A better indicator is the decoder itself. Instead of returning a moving average of CRC failures, the BER monitor returns a moving average of the number of packets which have been corrected by the decoder. In the case of a convolutional code (*i.e.* Viterbi decoder on the receiver), a non-zero accumulated Hamming distance for the selected path is a good indicator that the packet has incurred an error[LC83].

On the top of the FPGA implemented BER monitor, we run a software implemented error counter which polls the BER monitor, reads the $s[k]$ value of equation (4.1) and increases or decreases the counter depending on $s[k]$. Alternatively, it can be viewed as a state machine shown in Figure 4.9, whose state moves to the left or right depending on the $s[k]$.

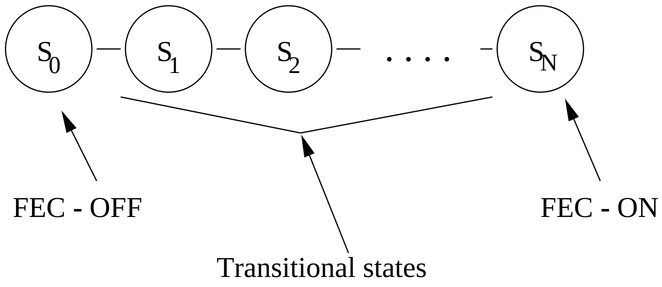

Figure 4.9: State machine representation of an error counter

Initially, the counter is set to zero (state machine is in its leftmost state) and the FEC is not engaged. If the value read from the BER monitor is greater than zero, the counter increments by this value (state machine advances to the right). If the counter reaches its maximum (rightmost state), the FEC is inserted. The counter may not count above its maximum nor below zero. Reading zero from the BER monitor, will decrement the counter by 1 (move the state to the left). When the counter reaches zero, FEC is removed from the protocol stack. In all intermediate states the state of the protocol graph is unchanged. The error counter (denoted as $C[k] \in \{0 \ldots N\}$) behavior can be mathematically expressed as:

$$C[k+1] = \begin{cases} 0 & z[k] < 0 \\ N & z[k] > N \\ z[k] & otherwise \end{cases}$$

$$z[k] = \begin{cases} C[k] + s[k] & s[k] > 0 \\ C[k] - 1 & otherwise \end{cases} \tag{4.2}$$

The error counter provides some inertia to the system as multiple consecutive non-zero reads from the BER monitor are necessary to insert the FEC. Informally, the policy module must be "convinced" that the error rate has indeed become sufficiently high before it decides to insert the FEC mechanism. The same rationale is followed when removing the FEC.

4.3.1 Tunable Parameters

Our policy module can be tuned to a particular error model with three parameters:

1. Moving average window size in packets (W)
2. Counter maximum value in error events (N)
3. Polling period in seconds (T)

The moving average window size (W) determines the bit error rate for which the FEC will be turned on or off. We call this bit error rate the *critical BER*. If the expected value of reading from the BER monitor ($s[k]$) is greater than 1/2, the policy module will eventually converge to the rightmost state (Figure 4.9) and initiate the FEC mechanism module insertion.

Hence, for a bit error rate greater than the critical BER, the expectation of $s[k]$ must be greater than 1/2. From equation (4.1) and linearity of an $E\{.\}$ operator, the requirement can be written as:

$$E\{s[k]\} = \sum_{i=0}^{W-1} E\{r[k-i]\} > \frac{1}{2} \tag{4.3}$$

The packet error indicator, $r[k]$, is a Bernoulli random variable[3] and its expectation equals the probability of a packet error at time k. If the bit error rate changes slowly, we can assume that the error probabilities for all $r[k]$ inside the moving average window are almost equally distributed. Given this assumption and the packet length of L bits, condition (4.3) becomes:

$$W > \frac{1}{2 \cdot L \cdot P_{bit}} \tag{4.4}$$

The inequality (4.4) allows us to determine the window size, W, so that the FEC insertion is guaranteed when the bit error rate reaches P_{bit}. A similar inequality, with the opposite sign, can be derived for the window size guaranteeing the FEC removal.

[3]For a discussion on the relationship between the exponentially distributed error interarrival time generated by the Network Impairment Module and the packet error rate, see Appendix A.

We have experimentally verified this analysis by varying the window size W and observing the behavior of the policy module. Results for a 2048 byte packet are shown in the Table 4.2 and plotted in the Figure 4.10 and Figure 4.11. Note that when the FEC is engaged, the effective packet size is 4096 bytes, because the FEC encoder adds one extra cell to each original cell.

The second column of the Table 4.2 ($P_{ON}^{predicted}$) shows bit error probabilities for which the policy module would insert the FEC mechanism according to equation (4.4). The BER for which the actual insertion has occurred is shown in the third column ($P_{ON}^{measured}$). The fourth and fifth column are the predicted and measured bit error rates for which the policy module will remove the previously inserted FEC mechanism.

Equation (4.4) specifies when the FEC insertion is *guaranteed.* It is possible that the actual insertion occurs earlier which explains the differences between measured and predicted critical BER. However, a good match in the order of magnitude validates the model of equation (4.4).

W [packets]	$P_{ON}^{predicted}$	$P_{ON}^{measured}$	$P_{OFF}^{predicted}$	$P_{OFF}^{measured}$
64	$4.8 \cdot 10^{-7}$	$1.0 \cdot 10^{-7}$	$2.4 \cdot 10^{-7}$	$4.2 \cdot 10^{-7}$
128	$2.4 \cdot 10^{-7}$	$8.3 \cdot 10^{-8}$	$1.2 \cdot 10^{-7}$	$2.1 \cdot 10^{-7}$
256	$1.2 \cdot 10^{-7}$	$\mathbf{6.3 \cdot 10^{-8}}$	$6.0 \cdot 10^{-8}$	$1.0 \cdot 10^{-7}$
512	$6.0 \cdot 10^{-8}$	$4.2 \cdot 10^{-8}$	$3.0 \cdot 10^{-8}$	$4.8 \cdot 10^{-8}$
768	$4.0 \cdot 10^{-8}$	$3.1 \cdot 10^{-8}$	$2.0 \cdot 10^{-8}$	$\mathbf{3.6 \cdot 10^{-8}}$
1024	$3.0 \cdot 10^{-8}$	$2.8 \cdot 10^{-8}$	$1.5 \cdot 10^{-8}$	$2.1 \cdot 10^{-8}$

Table 4.2: Predicted and measured critical bit error rates

From Figure 4.7, it can be observed that, in our example, the critical point is at BER $\approx 5 \cdot 10^{-8}$. It is also desirable to provide some hysteresis to avoid "thrashing" the protocol graph (*i.e.*, rapidly inserting and removing the mechanism module) when the system is operating near this critical point. Small fluctuations in bit error rate around the critical point will be damped by the hysteresis. Given the above result and assuming a packet size of 2048 bytes (4096 with FEC), we have chosen the window size of 256 when the FEC is not engaged and 768 when the FEC is in the protocol stack, which results in the hysteresis loop placed between $3.6 \cdot 10^{-8}$ and $6.3 \cdot 10^{-8}$.

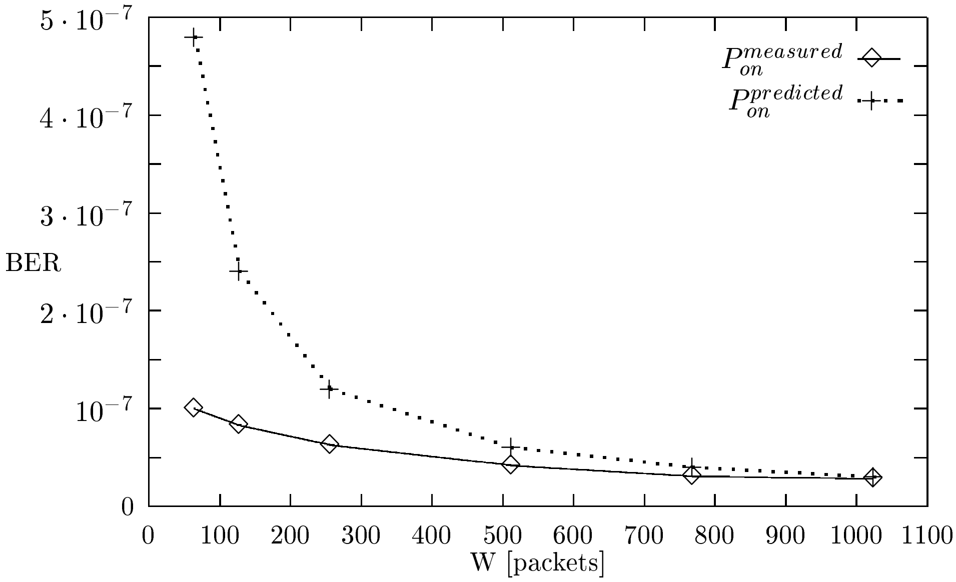

Figure 4.10: Predicted and measured bit error rates for which FEC insertion is guaranteed

The counter maximum, N, provides "inertia" for the system and can be determined by a tradeoff between the system speed and robustness. A larger N reduces the thrashing probability making the system more robust, but extends the system reaction time making the system slower. We show this tradeoff in the analysis to follow.

The policy module can be modeled as a Markov chain shown in Figure 4.12. The top subchain models transitions when the FEC is not engaged until its insertion, while the opposite transition is modeled with the bottom subchain. States $0 = 0'$ and $N = N'$ represent bounds of the counter. Two subchains are necessary because of different transition probabilities and chain lengths.

The transition probabilities $P(s)$ and $P'(s)$ represent the probabilities that the BER monitor will read s when the system is operating with and without FEC, respectively. They are defined as:

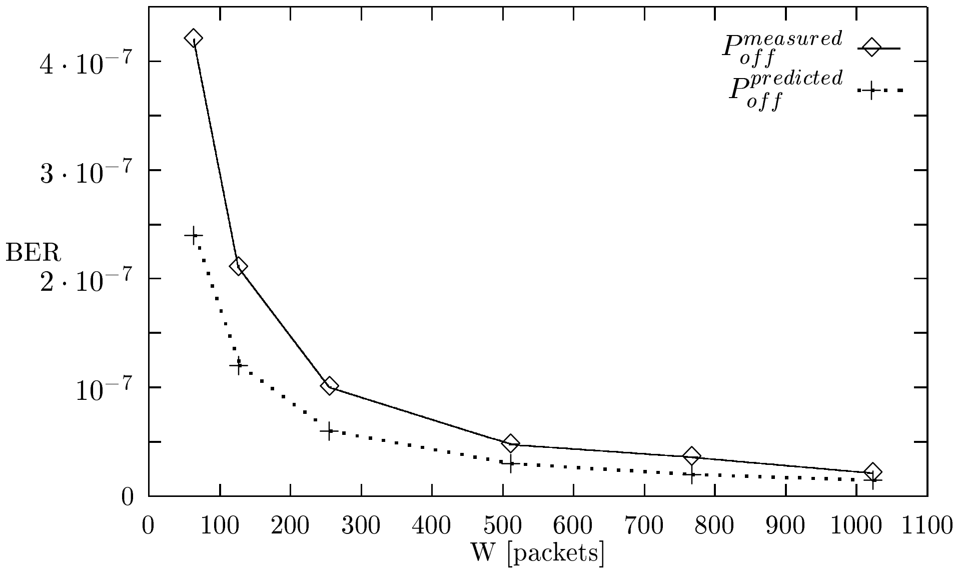

Figure 4.11: Predicted and measured bit error rates for which FEC removal is guaranteed

$$P(s) = p(s)\Big|_{W=256, L=2048\cdot 8}$$

$$P'(s) = p(s)\Big|_{W=768, L=4096\cdot 8} \tag{4.5}$$

$$p(s) = \begin{pmatrix} W \\ s \end{pmatrix} (P_{bit} \cdot L)^s \cdot (1 - P_{bit} \cdot L)^{W-s}$$

Since all the parameters necessary to fully describe a Markov chain have been specified, it is straightforward to numerically find the steady state solution vector $[q_i]$, where q_i is the probability of being in state i. Of special interest is the behavior of probability of being in state 0 and probability of being in state N. We call these probabilities q_{remove} and q_{insert} respectively, as they represent probabilities of removing and inserting a mechanism module (FEC in our example) into the protocol stack. Figure 4.13 shows the behavior of q_{remove} (dashed line) and q_{insert} (solid line) for different values of N and error rate around critical point. As expected, for an error rate well below the critical point q_{remove} dominates while

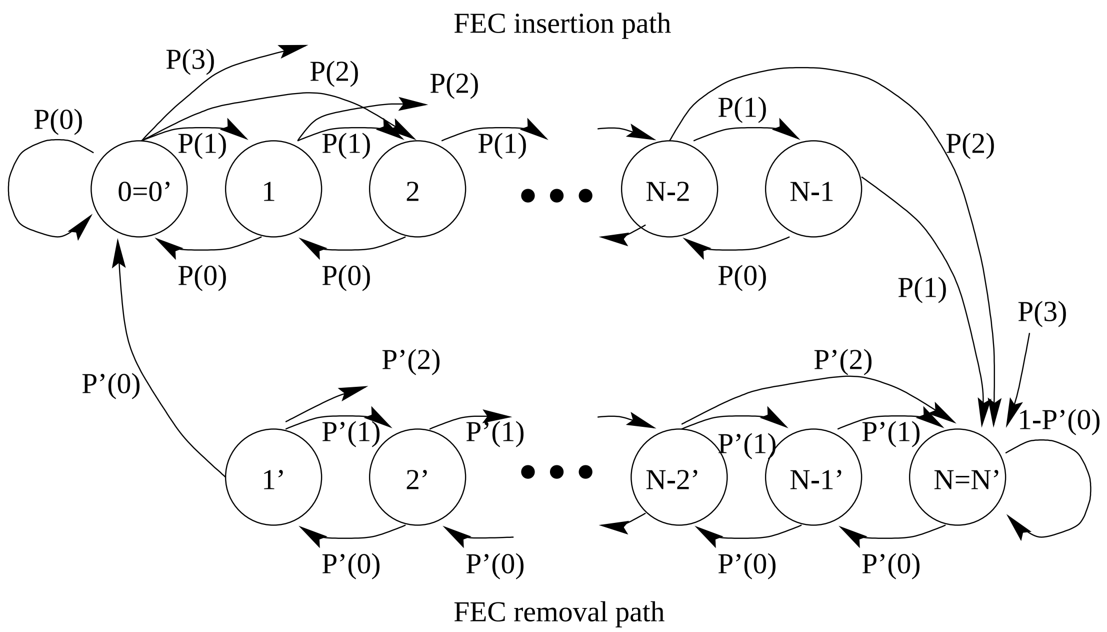

Figure 4.12: Markov model of a decision module

q_{insert} is negligible and vice-versa. There is also the uncertainty region in the vicinity of critical point where q_{remove} and q_{insert} are comparable. In this area the system is most susceptible to protocol graph thrashing.

It can be observed from Figure 4.13 that for larger N, the uncertainty region becomes narrower with the minor shift of the critical point to the left. A desirable property is to have the uncertainty region as narrow as possible, and the dominant effect of the counter maximum value N is to control the width of this region.

On the other hand, a large counter maximum value will make the system inert as more time will be needed to make the transition. As discussed earlier, some amount of inertia is necessary to filter out short events that may induce system oscillation. However, too much inertia could filter out some events that are otherwise addressable by the mechanism module. In Figure 4.14 we have plotted q_{remove} and q_{insert} as a function of time and error rate fixed to 10^{-7} (a BER for which it is clear that the FEC should be switched in). The plots are shown for several values of the counter maximum N and the influence of N on inertia is obvious. In our example we have chosen $N = 25$ as a trade-off between inertia and the width of the uncertainty region.

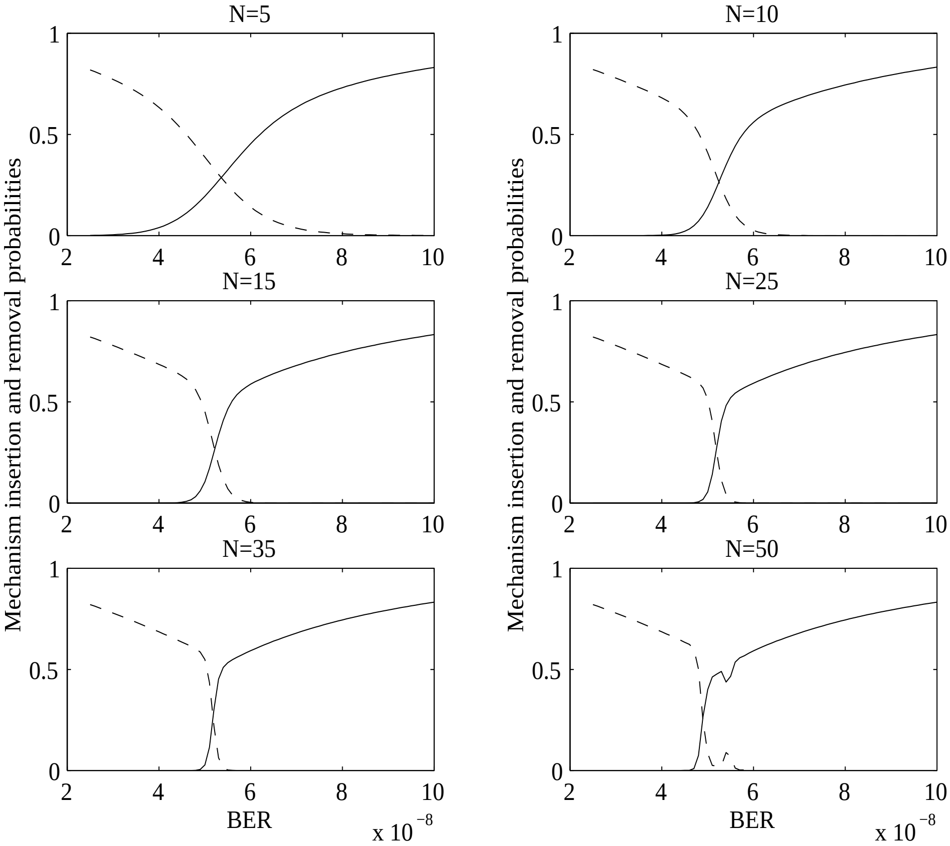

Figure 4.13: Steady state solutions: Probabilities of switching the mechanism module in (solid line) and out (dashed line) as a function of BER

The polling period, T, determines how often a processor reads data from the BER monitor. The system will operate faster with shorter polling period. Also, our assumption about slowly changing bit error rate favors small T. On the other hand, a short polling interval may result in overlapping moving average windows, which can affect the accuracy of models (*i.e.*, Markov chain model) we used to determine other tunable parameters. However, in our experiments, we observed a higher probability of oscillation with a shorter T suggesting that there is a lower limit on the polling interval. For an OC-3 bit rate, we have empirically found that for T=200 ms, the probability of oscillation is sufficiently small and that our models still give accurate predictions.

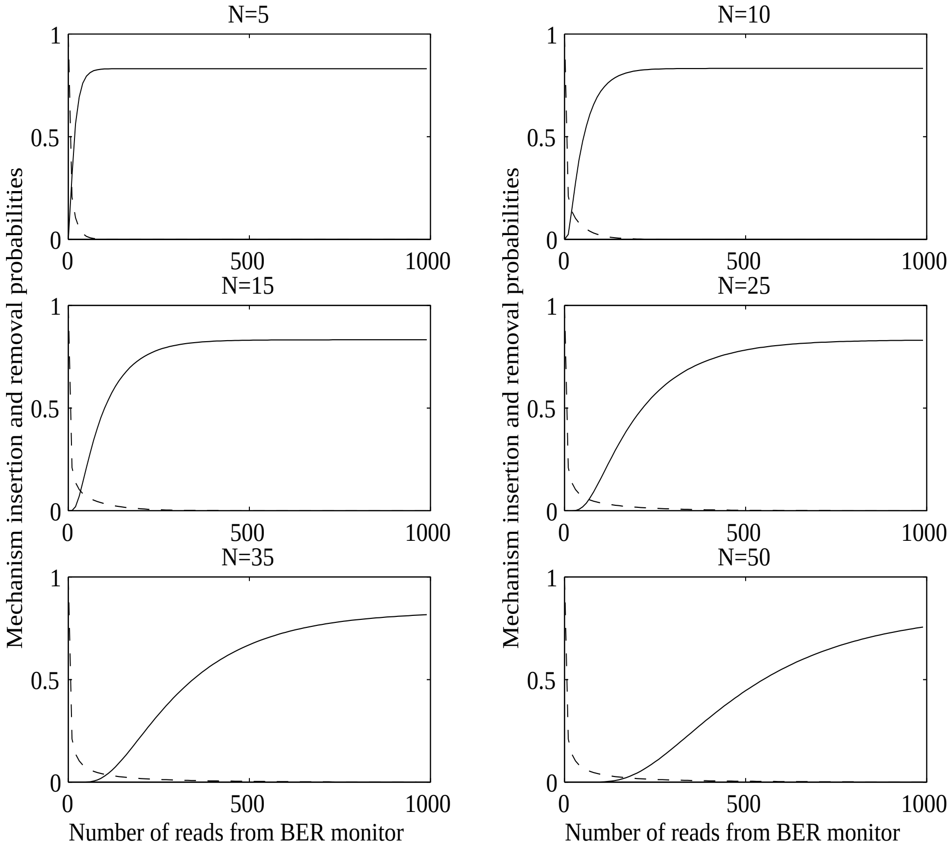

Figure 4.14: Transitional behavior: Probabilities of switching the mechanism module in (solid line) and out (dashed line) as a function of time

4.4 Signalling Protocol

Forward Error Correction is a distributed mechanism: it consists of an encoder at the transmitter side and a decoder at the receiver. Similarly, many other mechanism modules in the Protocol Boosters model are distributed consisting of a booster and a debooster. For efficient implementation, the booster and debooster must be inserted into the protocol stack synchronously, and a signalling protocol is necessary to provide this synchronization.

We now illustrate how the signalling protocol synchronizes the insertion and removal of the FEC encoder and decoder. Detailed specification of our signalling protocol can be found in [Had98].

The left side of Figure 4.15 shows an imaginary protocol stack (TCP/IP on top of ATM in our case study) without the booster (*e.g.*, FEC) in place. On the right side the same protocol is augmented with a booster at the transmitter side and a debooster at the receiver side. The application is not aware of modifications in the underlying protocol stack except for the performance improvement. The goal of a signalling protocol is to enable the transition from the state shown on the left side to the state shown on the right side without disrupting the communication.

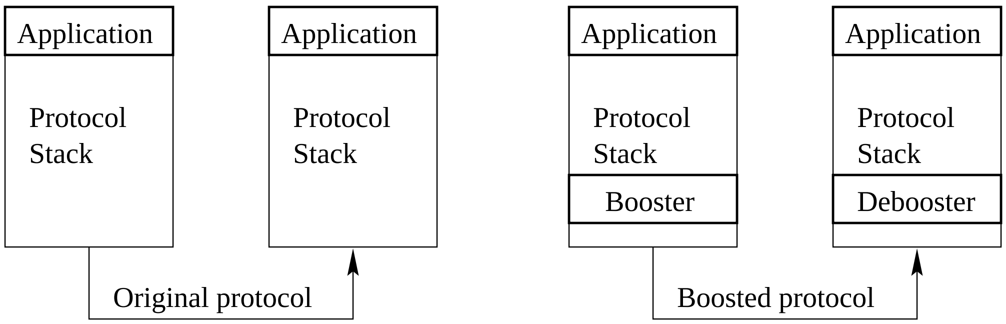

Figure 4.15: Original protocol (left) and the protocol modified by inserting a booster (right)

In our implementation, we embed signalling messages into a data stream by transmitting them over the same virtual channel with cell type fields set to binary 111 (the reserved OAM cell). The transmit section on the P4 network interface can generate signalling cells at the controller's request, while the receive section passes signalling cells to the controller. The signalling protocol is implemented in software and each booster or debooster has a dedicated thread executing the protocol.

There are two classes of signalling messages: messages used to control booster insertion and removal and messages used to deploy new boosters. In this section we discuss the first class while the second class is covered in the next chapter (in Section 5.4) where we discuss adding end-user programmability.

The signalling message carries the command code, the source ID, used to identify the source booster and the destination booster ID (all 8-bits long). The command code identifies the class of message and the command within the class. The CRC is added for error protection. The signalling message is transmitted in the first four bytes of the signalling cell payload and the remaining octets of the cell are unused. The P4 calculates the CRC in hardware prior to transmission and verifies it at the reception. Signalling cells that fail the CRC check are silently dropped by the receive section of the network interface.

We now illustrate the insertion and removal of the FEC booster described earlier in this chapter. The exchange of signalling messages is shown in Figure 4.16.

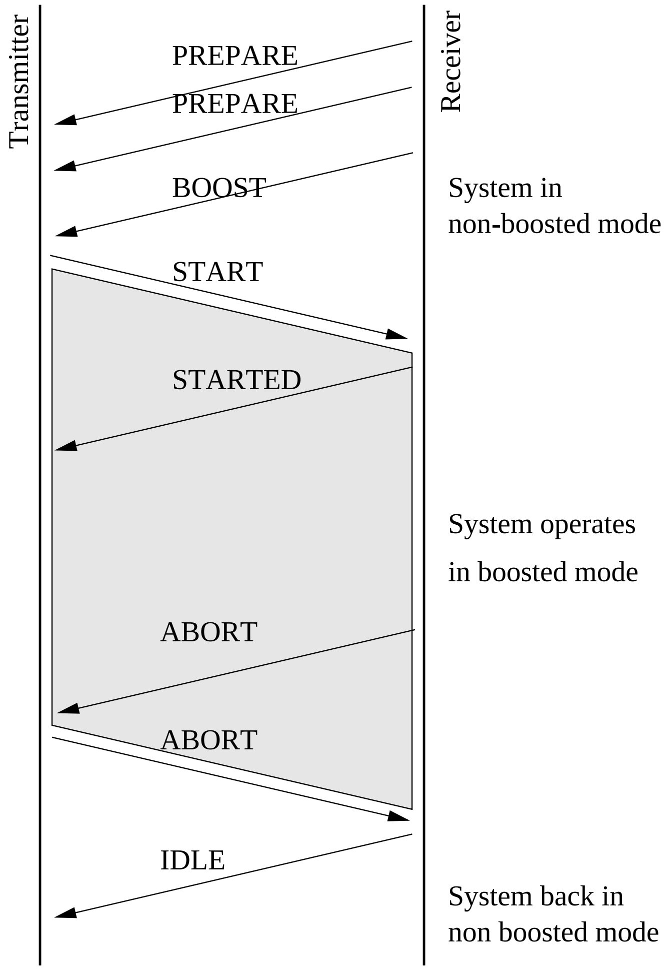

Figure 4.16: Handshake Procedure

At the receiver side a thread associated with the decoder wakes up at timer heartbeats, reads the value from the BER monitor and updates the error counter. As long as the error counter remains zero, no signalling messages will be exchanged. Meanwhile, the transmitter side will indefinitely block the thread associated with the encoder because the transmitter does not have access to the information about bit errors. Hence, the receiver makes the decision of when to insert the mechanism (*i.e.*, the policy module is located at the receiver). In other cases, the opposite might be true, or any side could be allowed to make policy decisions. The support for transmitter driven decisions is provided by the signalling protocol and the system knows which side can make decisions by the existence or non-existence of the appropriate policy module. In the FEC case, the debooster consists of a decoder, BER monitor and the software implemented error counter which makes decisions while the booster consists of an encoder only.

If the error counter becomes greater than zero, the policy module anticipates that the booster might be needed soon and sends the `PREPARE` message to the other side. An FPGA device on the P4 board will be configured on both sides: the transmitter will download the encoder driven by the arrival of the `PREPARE` message, while the receiver will download the decoder driven by the non-zero value of the error counter. However, the processing elements on both sides will be excluded from the pipeline and the protocol will still run without the FEC. Configuring the FPGA is a time consuming operation, and downloading in advance enhances the agility of the system by ensuring that the device will be ready before it is actually needed. The receiver side will keep sending the `PREPARE` message at every timer tick as long as the error counter is greater than zero. If a message is lost, the new message will be sent at the next tick eliminating the need to acknowledge `PREPARE` messages. The transmitter side will check if there is an FPGA device loaded with the encoder at each arrival of the `PREPARE` message, so the system will be maintained in the alerted state.

If the bit error rate is sufficiently high, the error counter will converge to its maximum and the policy module will decide to insert the mechanism. The `BOOST` message is sent to the transmitter side. After it receives the first `BOOST` message, the transmitter will insert the encoder and send the `START` message marking the beginning of the encoded cell stream. The receiver will start decoding after it receives the `START` message and respond with `STARTED`. The `BOOST` message is sent periodically similarly to the `PREPARE` message until the arrival of `START` message. The `START` message is also sent periodically until it is acknowledged with the `STARTED` message. After the modification of the protocol graph has been completed, no further signalling messages will be exchanged. Hence, the overhead of the signalling protocol exists only in the transitional state.

Loss of any signalling message except `START` will only delay the booster insertion but will not disrupt the communication. If the `START` message is lost, the encoder will be inserted on the transmitter side while the receive side will operate without the decoder resulting in a communication outage. However, at the next timeout, the new `START` message will be sent which will initiate the decoder insertion and the protocol stacks will be matched again. In our experiments, the outage was never long enough to seriously affect the ongoing communication and the TCP throughput always fully recovered after such outages.

The booster is removed when the error counter falls back to zero. Since there is no preloading involved, the message analogous to `PREPARE` is not necessary. Instead, the receiver side sends the `ABORT` message and awaits the transmitter's response. The transmitter sends the `ABORT` message in response marking the end of the encoded data stream and removes the encoder[4]. The receiver stops decoding the cell stream when it receives the `ABORT` message and responds with the `IDLE` message. Loss of the `ABORT` message from the transmitter to the receiver will result in a communication outage, but the system will recover after the next `ABORT` message is retransmitted.

Although any side is allowed to make policy decisions, only the transmitter side can mark the point in time at which the boosting starts. This is a general rule for all boosters. Allowing the receiver to start deboosting first would result in a period of protocol mismatch at every transition. With this restriction, a protocol mismatch can happen only if the `START` or the `ABORT` messages are lost making it an exception rather than a rule.

4.5 Experimental Evaluation of the Policy Module

Static measurements of Section 4.2 evaluate the mechanism module. So far, we have experimentally shown that the opportunity for adaptive protocol indeed exists in our case study and that the hardware implementation is essential to achieve acceptable performance. We also described the policy module whose performance we now evaluate.

The fundamental question is if and by what amount the overall system performance is better when the FEC booster is controlled by the policy module compared to the non-adaptive system. There are two non-adaptive systems we use as a baseline:

1. Unmodified TCP.

[4]Note that the receiver's `ABORT` messages only requests the booster removal, while the transmitter's `ABORT` message actually marks the removal point.

2. TCP with FEC statically inserted.

4.5.1 Performance Indicator

We define the *boost factor* as the ratio between the performance of the adaptive system and the baseline performance:

$$b = \frac{\eta_{adaptive}}{\eta_{static}} \tag{4.6}$$

Values necessary to calculate the boost factor can be easily measured by setting the Network Impairment Emulator to change the bit error rate over time following some previously defined pattern and measuring the throughput with `ttcp` test tool over a long period. The throughput of an adaptive system and the two non-adaptive baseline systems is measured and the boost factor is calculated according to the equation (4.6). If the boost factor is greater than 1, the adaptive system performs better and the greater value indicates better performance improvement.

4.5.2 Results

First, we measured the boost factor as a function of the bit error rate for the case where the conditions on the link periodically change between the error-free and noisy state. A real-world example analogous to this scenario would be the behavior of a satellite link in the face of changing weather conditions or a man-made noise induced in cable-modem systems. For simplicity, we wanted to minimize the number of system parameters and therefore decided to use the channel whose conditions vary according to some deterministic pattern. If this system were used in a real-world application, similar measurements should be done for a link whose conditions change randomly according to some statistics that describe the channel. Since our only goal was to demonstrate how the adaptive protocol reacts to the changes in network conditions and improves the overall performance, a deterministic model was sufficient.

We programmed the Network Impairment Emulator to keep the link error free for time T_0 and then switch to the error-generating mode with the bit error rate set to BER_0 and keep it for time T_0 after which the process repeats as shown in Figure 4.17.

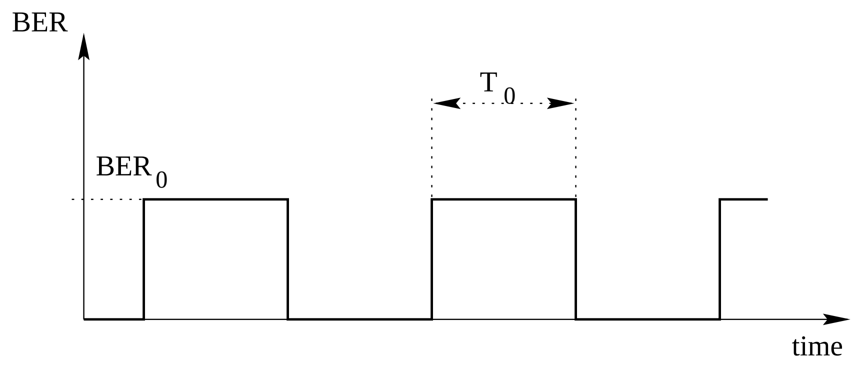

Figure 4.17: Bit error behavior during the first set of measurements

The `ttcp` test runs with a very long data stream (`-n 1000000`) so that the total duration of a connection is always large compared to T_0 resulting in many transitions from error free to error mode. The BER in the errored intervals was fixed to BER_0 throughout the entire measurement session; multiple measurements were made for each BER_0 and for each T_0, and the mean was calculated.

Figure 4.18 shows the boost factor with respect to the throughput of unmodified TCP protocol stack (*i.e.*, the reference throughput η_{static} was measured without FEC). We measured the boost factor for two different values of T_0: 20 and 40 seconds.

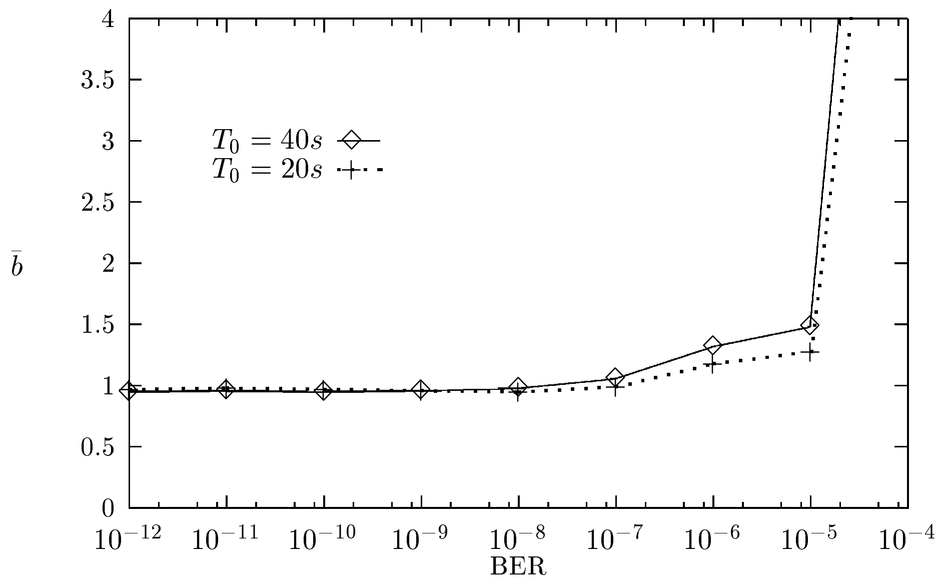

Figure 4.18: Boost factor with respect to the throughput of unmodified TCP (function of BER)

For low bit error rates, boost factor is approximately 1 or slightly below 1, because the FEC is never engaged and the system acts as unmodified TCP protocol. A small difference in favor of the static protocol stack is a consequence of the overhead for signalling messages. The booster insertion never occurs because the error counter never reaches its maximum, but bit errors cause the `PREPARE` message to be sent over the communication channel resulting in a small overhead. The improvement rapidly grows with the bit error rate and becomes practically infinite for $BER > 10^{-4}$. In the BER region above 10^{-4}, the unmodified TCP closes its congestion window and almost completely stalls with insufficient time to recover when the link returns to the error-free state. The adaptive protocol inserts the FEC booster, preventing the TCP from ceasing to function, and removes the FEC booster when the link goes back to the error-free state.

Figure 4.19 shows the boost factor with respect to the throughput of a TCP protocol stack with the statically instantiated FEC. To determine the reference throughput for this case, we manually forced the FEC insertion before the test session and ran the throughput test with the policy module disabled.

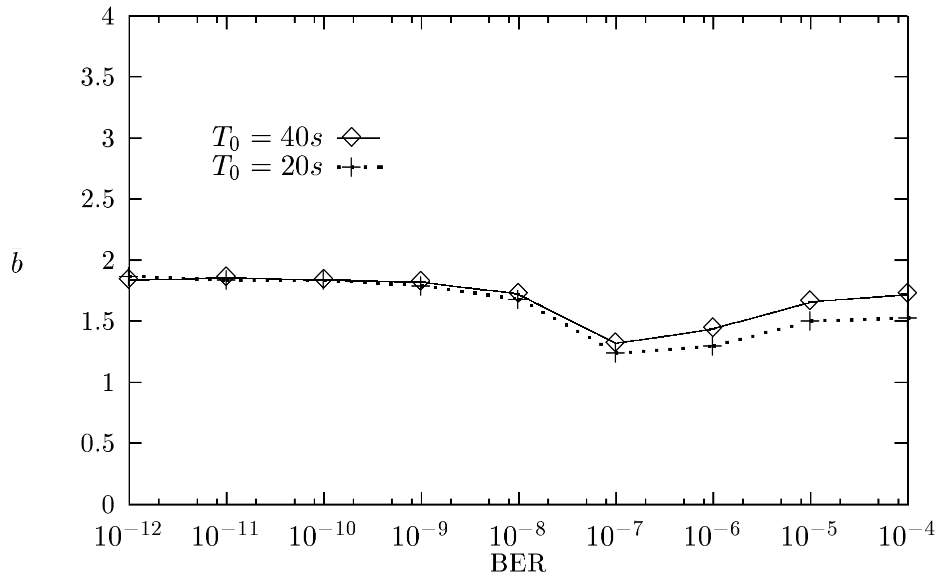

Figure 4.19: Boost factor with respect to the throughput of TCP with static FEC (function of BER)

Improvement occurs for both low and high bit error rates. In the low BER regime the overhead of statically instantiated FEC prevents the TCP from fully opening the congestion window and almost 100% improvement is due to rate control at the transmitter. For high bit error rates, the static FEC improves the TCP throughput when the link is in the error mode, but does not take the advantage of error-free periods. Again, the adaptive protocol performs better by removing the FEC when it is not necessary.

In the second set of measurements we fixed the bit error rate and programmed the Network Impairment Emulator to insert the bit errors for time T_1 and provide the error free channel for time T_2, where $T_1 + T_2$ was fixed to 120s, while T_1 and T_2 were varied. In other words, we varied the duty cycle of the graph shown in the Figure 4.20.

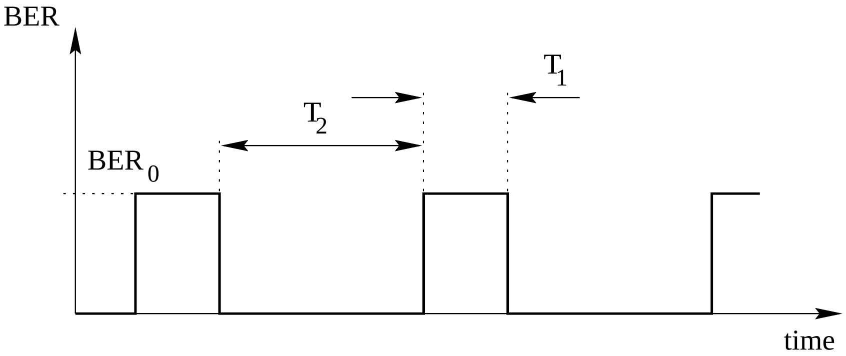

Figure 4.20: Bit error behavior during the second set of measurements

The second set of measurements relates the throughput improvement with the percentage of the time when the channel in noisy. Figures 4.21 and 4.22 show the boost factor as a function of the duty cycle, d, defined as:

$$d = \frac{T_1}{T_1 + T_2} \tag{4.7}$$

The zero value of the duty cycle represents an error free channel, while 1 represents the channel constantly in error mode. For $d = 0.5$, we get the set of measurements for scenario shown in the Figure 4.17 with $T_0 = 60s$.

In the first case (Figure 4.21), a significant throughput improvement can be observed as d approaches 1. This is because the FEC stays inserted longer and the protocol stack benefits more. As noisy periods become shorter, the adaptive protocol uses the FEC less frequently and the benefit decreases. Ultimately, when there are no error periods, the

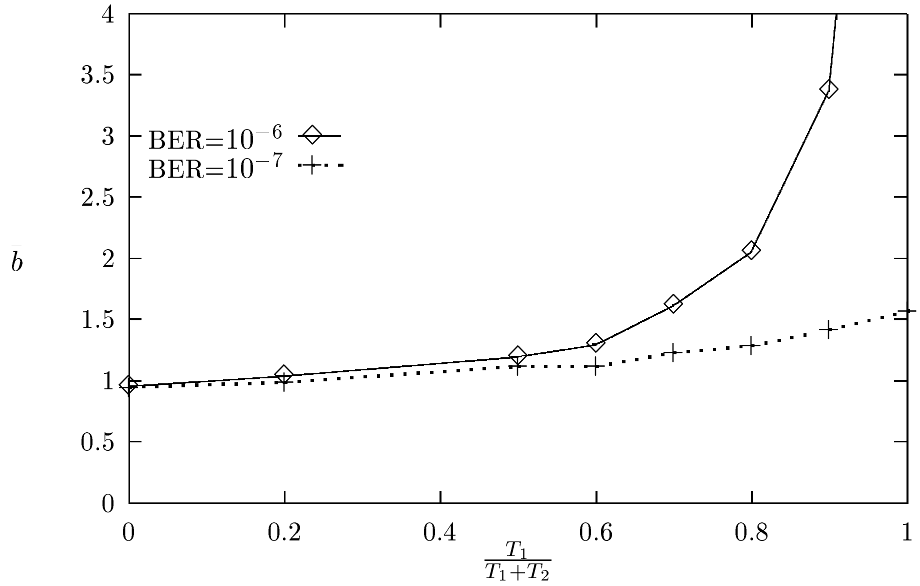

Figure 4.21: Boost factor with respect to an unmodified TCP (function of duty cycle)

adaptive protocol converges to the original plain TCP and the boost factor is approximately 1. When compared to the TCP with statically inserted FEC, the adaptive protocol performs better when the error periods are short, taking the advantage of its ability to remove the FEC.

To summarize the discussion so far, we have designed and implemented a system (hardware and software) and used an example to experimentally demonstrate that the protocol can indeed benefit from the adaptation, and that hardware implementation is essential for certain class of protocol processing functions. The experiments have shown that there exists an operating region in which the adaptive protocol outperforms the statically constructed protocol while in other regions the adaptive system imposes small or no performance overhead. Finally, comparison with the software system justifies our concept of using reconfigurable hardware to build the reconfigurable network.

In the remaining parts of this chapter we provide a general framework for implementing our concepts without limiting the discussion to forward error correction.

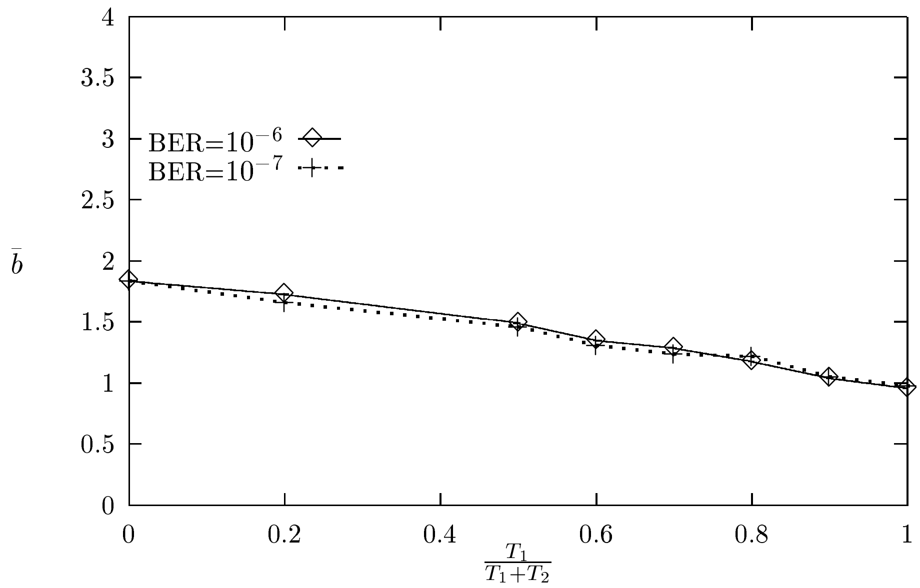

Figure 4.22: Boost factor with respect to the throughput of TCP with static FEC (function of duty cycle)

4.6 Generalizing Adaptive FEC in Hardware

The P4 demonstrates near-software flexibility and performance comparable to special purpose hardware. We used a convolutional code as a mechanism example for an FEC Booster. Using the FEC adaptively has been studied by other researchers in the past. One such scheme proposed by Lin and Yu has been known as "Type II Hybrid ARQ"[LY82][LY83]. In this scheme, the transmitter sends the user data along with the parity bits necessary only to *detect* the error. If an error is detected the receiver requests the retransmission, but instead of retransmitting the whole packet, the transmitter sends only the additional parity bits necessary to correct the error. Since the number of parity bits is usually smaller than the packet size, the bandwidth is used more efficiently than if the packet were nonselectively retransmitted. The number of retransmitted parity bits may be increased, making the code stronger, until it ultimately reaches the size of the packet when the bandwidth utilization equals that of a simple retransmission. The FEC is adaptive in the sense that the *coding rate* is changed to accommodate the current link characteristics. In contrast to our approach, in Hybrid ARQ scheme the FEC module must always be inserted in the protocol stack and the parity bits must always be calculated (but not necessarily transmitted). The result is a static protocol with limited adaptive behavior. Also Hybrid ARQ can be used for only certain coding schemes, where our approach is not dependent on the particular code.

Another example of an adaptive FEC which works in conjunction with ARQ has been presented in [Say96]. Based on received acknowledgements the transmitter estimates the link error rate and attempts to use the best coding rate. The receiver attempts to decode the message using all available decoders in parallel. The one with the smallest coding rate which does not fail the CRC check after the decoding is assumed to be the inverse of the encoder used by the transmitter. Due to the lack of synchronization between the transmitter and the receiver, the receiver must have all available FEC modules in the protocol stack and attempt decoding using each one of them. In the general boosters framework discussed in the Section 4.7, the concept presented in [Say96] can be viewed as a policy module with missing synchronization function, and the need to try each decoder is a direct consequence.

There are two generalizations which distinguish our example from the adaptive FEC studied by other researchers. First, the mechanism is neither limited to a particular coding rate nor a single coding algorithm. Sets of codes appropriate for different BERs or burst lengths can be made available, and the best code for current conditions in the network can be selected by a policy. If conditions change, the code can be adapted at run time by reconfiguring the P4. Second, the P4 architecture can be used for much more than FEC. Any functional element (*e.g.*, encryption, data compression, traffic shaping modules, *etc.*) implementable on the P4 can be used as a protocol booster and added to a protocol stack on an as-needed basis.

4.7 General Discussion on Policy

The FEC booster example and its policy module allows us to identify the functions common to all policy modules:

1. *Measurements*: At the lowest level, there is a monitor module which collects the data from the network. It highly depends on the associated mechanism and the application. In our case study, this is the FPGA implemented BER monitor.

2. *Decisions*: Making policy decisions involves processing the measured data and deciding if the booster should be inserted, removed, or if the protocol graph should remain unchanged. For example, the error counter associated with our FEC booster maps the data collected from the BER monitor to a decision.

3. *Synchronization*: Synchronizing the distributed boosters is a function of the signalling protocol. In contrast to measurements and decision, the signalling protocol is neither booster nor application dependent.

We call the modules that perform the above mentioned functions the *monitor* , *decision module* and *signalling protocol*, respectively.

The following subsections describe each module in greater detail. Figure 4.23 shows the relationship between modules constituting a booster. The monitor, decision module and signalling protocol make up, the control plane of a booster. The mechanism module implements the data plane.

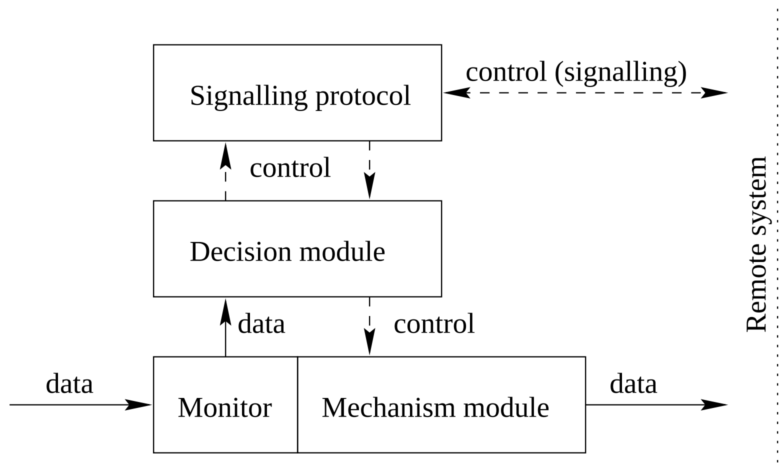

Figure 4.23: The structure of a booster (mechanism and policy module)

4.7.1 Measurements

When designing a monitor module, it is necessary to identify a set of relevant measurable parameters that will be used to detect a need for protocol adaptation. These parameters can range from something as simple as explicit requests (*i.e.*, messages) from the user or other policy modules to measurable quantities such as checksum errors, buffer utilization, signal-to-noise ratio (SNR), etc. Measurements can be taken at different layers of a protocol stack and at multiple points in the network (*i.e.*, measurements may be distributed).

A common property for all monitors is that the information collected is highly dependent on the associated mechanism and that the monitor must be designed in conjunction with the mechanism. The separation between the monitor and the mechanism module is sometimes context-dependent, and may be blurred.

4.7.2 Decisions

Information collected by the monitor module can be of any nature. This data must be processed and transformed into a result that represents a decision. In other words, a decision module maps the set of monitor outputs to a set of actions.

In the Protocol Booster model, the decision can be to insert or to remove the mechanism module. For convenience, we have added a third decision: remain in the current state. In other words, the set of actions is a three element set: { *insert*, *remove*, *continue* }.

The decision module is a clear separation between mechanism-specific parts and the rest of a policy module. Once a decision has been made, enforcing the mechanism insertion and removal as well as synchronization (if necessary) with protocol stacks on other network nodes is independent of the particular mechanism.

A decision module must be aware of the time needed to instantiate the new functional module. In the Protocol Boosters model, this time may include kernel loadable module instantiation time for boosters implemented in software[MMR98], device downloading time for FPGA implemented boosters[HS97], propagation time over a high latency network, etc. If the event that would trigger the modification of a protocol graph is shorter than the mechanism instantiation time, switching the module in would result in a mismatch between current network conditions and protocol instantiation. This is obviously undesirable, and the policy module should avoid reacting to such events. In our case study, we used the maximum value of the error counter to tune the inertia of the system and filter out short error bursts that would otherwise trigger FEC insertion after the burst has passed. The system inertia was set so that the probability of protocol stack modification is negligible for events shorter than the sum of the time to download the FPGA device and the signalling message propagation time. Figure 4.14 was used to relate this probability with the counter maximum value.

4.7.3 Synchronization

For distributed processing consisting of a booster–debooster pair, some form of explicit or implicit signalling is required. Many protocol processing functions have this property: encryption–decryption, compression–decompression, etc.

The role of a signalling protocol is twofold. Firstly, it must ensure that distributed functions are activated synchronously to avoid mismatched protocol stacks at the transmitter and the receiver.

Secondly, the signalling protocol allows distributed decisions, since there may be information asymmetry. We presented the example where only the receiver has the information about network characteristics and therefore a monitor module must be located at the receiver side. However, the transmitter must mark the point at which the boosting starts so that the receiver can start deboosting at the correct point in time. Placing the decision module at the transmitter may be impractical due to the cost of transferring measurement data across the network. A better solution, in general, is to place the decision module at the receiver and use the signalling protocol to transfer decisions across the network. Decision information may be sent only when the protocol state needs to be changed, which results in much less overhead traffic.

4.8 Software Architecture

After qualitatively and quantitatively studying an example booster, we now have the prerequisites to describe the architecture of the software that runs on the P4's controller.

Although this section logically belongs in Chapter 3, we believe that it is better to explain the software modules after the functionality of a booster has been understood on a concrete example.

At this stage we ignore all architectural elements that provide user-level programmability. We will return to the software architecture and discuss it in the context of programmability in Chapter 5. Here we assume the original Feldmeier Protocol Boosters[FMS+98] model and explain the interaction between controller's software and P4 hardware in supporting the model.

Figure 4.24 shows the high-level software architecture. The device driver provides the low level interface to the P4 board. The FPGA devices are downloaded by transferring the configuration from the main controller's memory to the device driver using a `write` system call. All other control functions are implemented through the universal `ioctl` system call. The device driver can control up to four P4 boards mapped onto same I/O addresses and multiplex them using the board identifier and separate interrupt request lines.

The booster and debooster are treated as separate entities in the controller's software. For simplicity, in this section, we will use the term "booster" to refer to either a booster or a debooster.

Each booster known to the system has an associated control thread and a booster descriptor data entry. Each control thread executes the same code (which includes the signalling protocol), except for the decision module which is booster specific. A field in the booster descriptor points to the code of the associated decision module. There are at most two decision modules per booster as different modules may be executed depending on whether the booster is engaged or not. Also, a decision module may be shared among boosters. The `NULL` value of a decision module pointer indicates that in the current context, the booster cannot make policy decisions. For example, the FEC encoder has a `NULL` value for both pointers. The control thread wakes up periodically and calls the decision module. Based on the result, the policy is enforced and the signalling protocol is executed.

Decision module reads from the monitor which is an FPGA configuration downloaded into the P4 board. The FPGA configuration that implements a monitor is the part of the booster descriptor. Again, the same monitor may be shared among several boosters. The

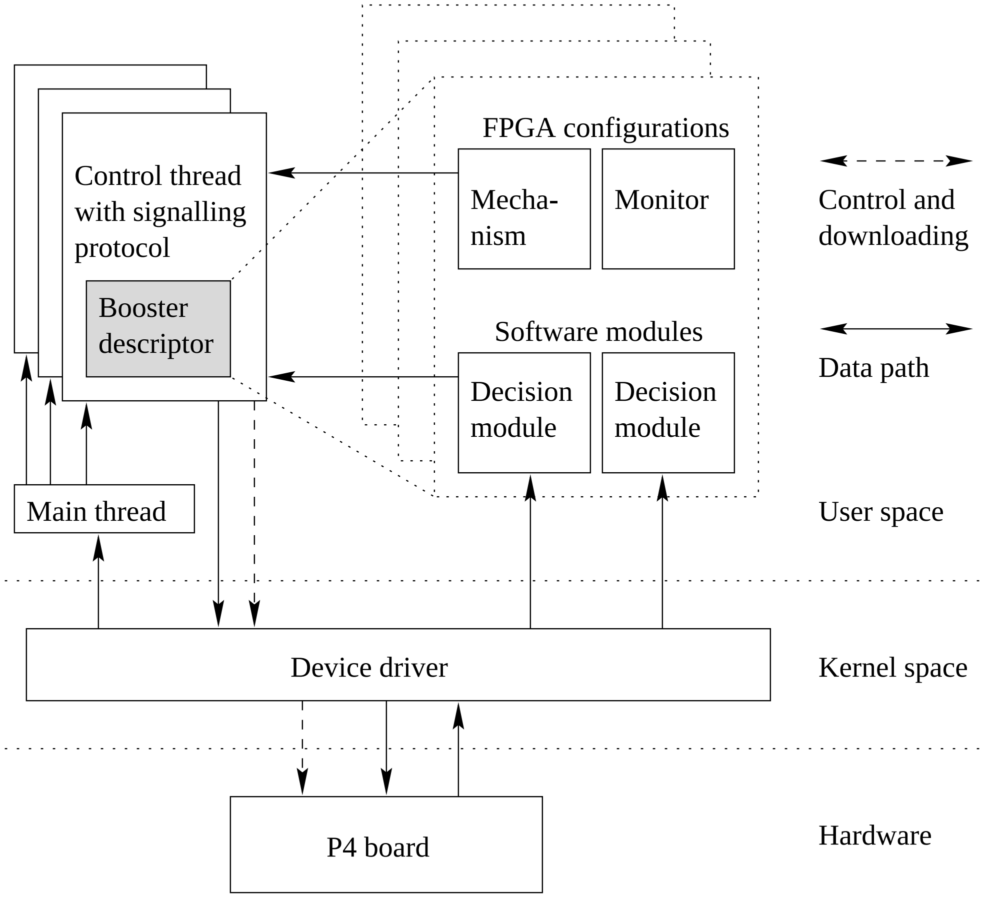

Figure 4.24: P4 software architecture

monitor is a simple and small configuration which does not modify a data stream (*i.e.*, it is not a booster). It is used only when the booster is not engaged. When the booster is in the protocol stack, the monitor must be implemented as the part of the booster (*i.e.*, in the same FPGA device).

The mechanism module is an FPGA configuration stored in the booster descriptor and downloaded into the P4 board prior to inserting a booster into the protocol stack.

The main thread demultiplexes signalling messages and delivers them to control threads. It waits on a `select` system call until the new signalling message arrives, reads the message from the device driver and passes it to the appropriate thread. After the signalling message is delivered, the control thread wakes up and responds to the message according to the signalling protocol. For efficient message multiplexing, each booster descriptor has the unique ID code of the associated booster and its counterpart (*i.e.*, debooster) to which messages should be sent. The signalling message carries the destination and the origin booster ID. The destination ID is used to deliver the message to a control thread, while the origin ID is used to check if the message has been sent by a valid counterpart.

Chapter 5

Support for Programmability

We have discussed the need, and technical foundations, for building adaptive protocols in hardware. We described an infrastructure that supports adaptation of hardware portions of the protocol stack and experimentally demonstrated its feasibility. We followed the Protocol Boosters model, whose main focus is adaptation, rather than programmability. In this chapter we focus on programmable network infrastructures, and apply the same concepts we learned from studying the adaptive protocols.

5.1 Augmenting Software Modules with Hardware

The basic concept we applied in Chapter 4 was augmenting the software system with dedicated flexible hardware. An example function, namely FEC, was implemented on P4 platform and used to provide a *service* to the original protocol stack.

The flow of data is illustrated in Figure 5.1. The left side of the figure shows the P4 case, while the right side is a generalization. In the generalized case, data processed by software portions of the protocol stack may be passed to hardware where a function is executed at high speed and passed back to software for further processing. This process may repeat several times before the data are actually transmitted.

The hardware modules serve as a fast path for protocol processing. If the protocol needs a function that is available in hardware, it can take the fast path and achieve higher

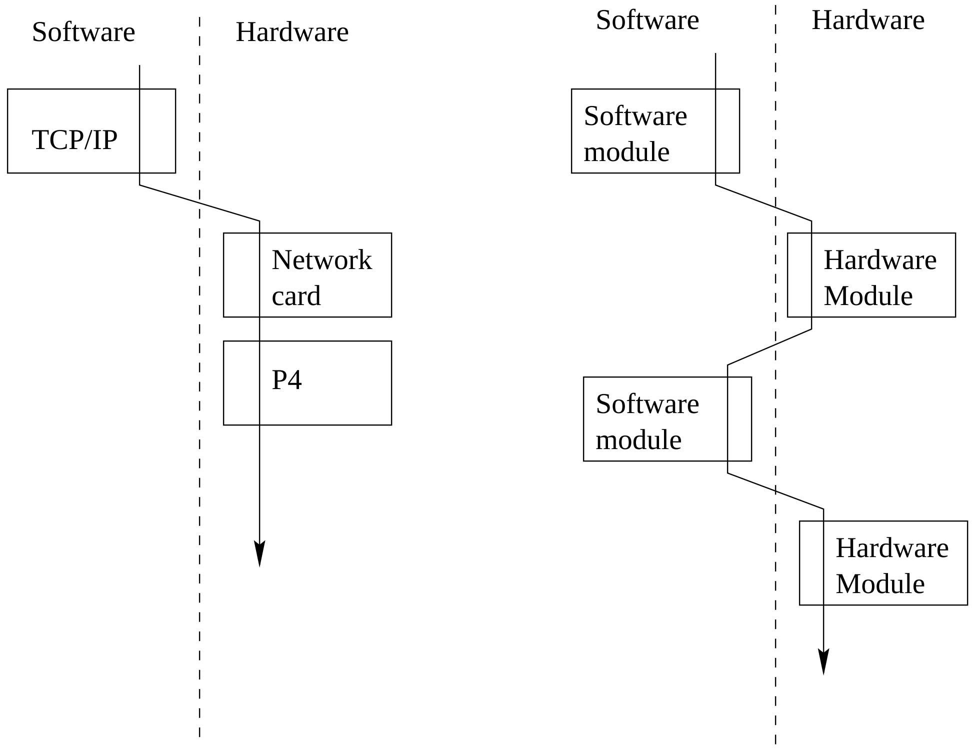

Figure 5.1: The flow of data in P4 augmented TCP/IP (left) and the flow of data in hardware augmented protocols in general (right)

performance than its software functional equivalent. Examples of non-programmable systems with such processing paths are communication systems with special purpose hardware for encryption[Cor99], checksum calculation[KSZ95] or systems with bus optimized for communications[Dal93], as well as switches or routers with most of the functionality implemented in ASICs and the software implemented control plane.

In the remaining parts of this chapter we discuss embedding programmability into hardware modules and providing the software support for such a programming model. Figure 5.2 shows the the evolutionary stages of embedding the programmability into a communication system.

The left side of the Figure 5.2 is a typical non-programmable network node (*e.g*, a portion of a switch or a host). The line card is by the definition implemented in hardware with little or no programmability. In general, protocol processing algorithms on the of

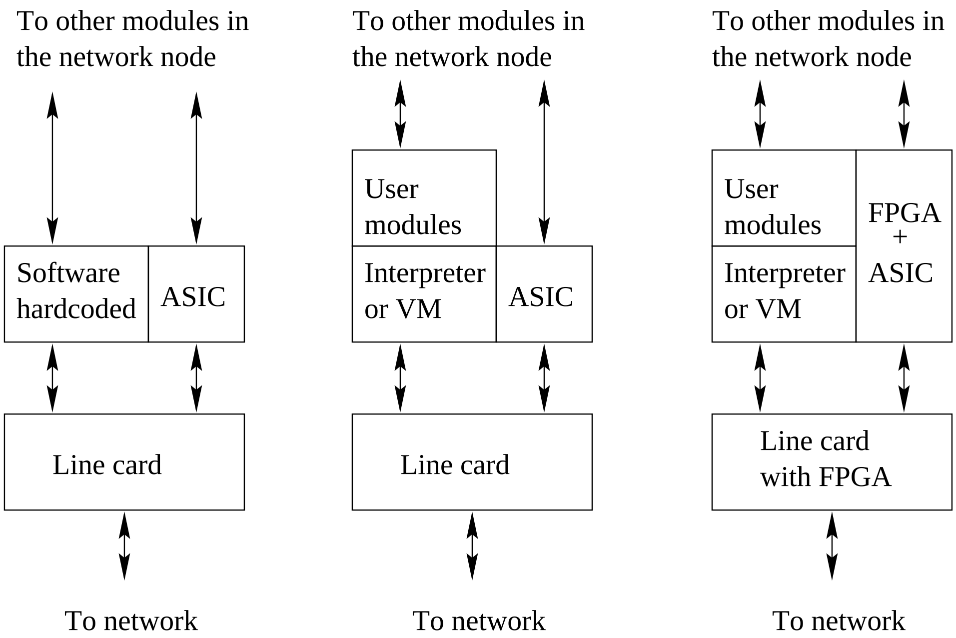

Figure 5.2: Programmable network node evolution

the line card are implemented as a combination of ASICs and software running on a general purpose processor. The ASIC is a hard wired circuit and by definition does not offer programmability. Software is inherently flexible, but in classical communication systems, typically written without any support for user-level programmability. Signalling provides limited ability to control the network behavior and some researchers have used it to provide the apparent programmability (*e.g.*, XBind project[LLM96]). By mapping network programs to signalling, the user has the impression of programming the network. Although this approach does not really program the network, its significance is in the ability to implement it on top of an existing commercial network infrastructure.

The middle part of the Figure 5.2 shows a typical programmable network node used in existing experimental networks[Smi99]. All support for programmability is in software portions of the network element, while the hardware is still a hard wired circuit. Examples are Java Virtual Machine[WGT98], PLAN interpreter[HKM$^+$98], CAML runtime system[ASNS97], etc. The node offers full programmability either by interpreting programs carried in the packet header, or by downloading protocol processing modules.

Finally, the right side of Figure 5.2 shows a node which offers full programmability of both hardware and software. The programmability of a line card allows the user to incorporate custom modules which process the packet contents as it goes through. The P4 architecture supports this concept and can be viewed as an extension to the processing path of a line card. At the network node, replacing ASICs with FPGAs provides a flexible architecture for implementing processing functions in hardware. For example, a board similar to the P4 with an added data path to the high speed I/O bus could be built to provide better integration of processing at the node and on the FPGA board. Using FPGAs has its drawbacks as ASICs can still deliver significantly higher performance. Programmable logic, therefore, provides a tradeoff between flexibility and complexity on one side and performance on the other.

5.2 Deploying User Defined Boosters

As described in chapters 3 and 4, the P4 extends the Protocol Boosters model forming an architecture which consists of a software protocol augmented by the programmable hardware. The straightforward transition from an adaptive system to a programmable system is to allow the user to define and implement the mechanism and the policy modules.

In our programming environment, the user implements the mechanism modules and deploys the FPGA configurations. The monitor module needed by the policy is also implemented as an FPGA configuration. Booster is deployed by transferring the FPGA configurations to the node and creating the appropriate booster descriptor. Corresponding policy modules are constructed from the monitor, the decision module and the signalling protocol. Signalling protocol is hard coded into the control software and common for all policy modules without user level programmability. Monitor module is an FPGA configuration written by the user with a well defined interface to the controlling software. Decision modules are implemented in software by the user, transferred to the node and dynamically linked with the control program. In general, the deployed booster consists of (typically) large FPGA configurations that implement the mechanism and the monitor module[1], and (typically) short decision modules[2].

[1] The size of the configuration file for the Altera 10K40 device that we use is approximately 64 kilobytes.
[2] In our experiments, decision modules were approximately 600 bytes long.

Extension towards programmability improves the network flexibility, but raises new concerns about the programming model and system security. We implement and distribute the decision modules using an approach similar to that of ANTS[WGT98] and Switchware[Ale98a]. A policy module is implemented in a type-safe and platform independent programming language. The byte-code is distributed and dynamically linked at the node. We have chosen CAML[Ler99] as the language for implementing decision modules. An extensive discussion on the choice of a language for programmable networks and comparison between CAML and other languages suitable for programmable networks has been presented in [Ale98]. In addition to that discussion, we note that using CAML is especially convenient for our application, as it allows the control program written in C to be linked with the CAML byte-code interpreter. This further allows the control program to invoke a decision module in the same address space using the CAML callback feature[Ler99]. The decision module can also invoke the C-implemented interface to a device driver, collect the results from the monitor and return the decision to the control program without the need to implement any inter-process communication structures.

The mechanism modules are implemented in VHDL and compiled for the architecture of the device used on P4 board. The raw binary configuration file is distributed. With the current state of the reconfigurable computing technology there is no analogy with the platform independent mobile software and no research that we are aware of has been done on this subject. Hence, the choice for deploying user defined mechanism modules was among the three options:

1. Distributing the source code in a hardware description language, such as VHDL.

2. Distributing a compiled, device dependent, FPGA configuration.

3. Distributing a netlist in an intermediate format, such as EDIF[Ele].

The main advantage of the first option is the ability to verify the VHDL code using results from the field of formal hardware verification[Mel93] enhancing the safety and security of the system. The major drawback is in the deployment latency since the VHDL source would have to be compiled at the node which is a costly operation. From our experience, the time to compile the VHDL code into an FPGA configuration is measured in minutes and sometimes even hours which makes source level distribution prohibitively slow. Theoretically, some level of platform independence could be achieved if the mechanisms were distributed in source, since the VHDL provides sufficiently high level of abstraction. In practice the device dependence is not completely eliminated by using VHDL since the resulting design performance highly depends on how the coding style matches the architecture. For example, register intensive device architectures prefer deeply pipelined designs and state machines with many states doing simple operations, while the look-up table intensive architectures prefer complementary design style.

The second option allows fast distribution of mechanism modules, but without the ability to verify the code. The raw binary configuration is analogous to an executable file in general purpose computing systems. The configuration is device dependent and cannot be verified for correctness. However the mechanism module is available for use immediately after downloading at the network node and the design can be highly optimized for performance which is the major advantage of this option.

The last option still requires compiling at the network node. The compilation is faster because the step that converts the VHDL source to netlist is eliminated, but the most costly step in compilation, "place and route", must be executed at the node. The ability to verify the design is lost and there is more device dependence than in the first case but less then in the second case.

We have chosen to distribute the compiled FPGA configurations, since we believed that the deployment latency is an important issue. All P4 boards in the system utilize the same device type, so device dependence is a secondary issue in our system and since the compiler is proprietary software whose source code is not available, there are technical difficulties with integrating a compiler with the node resident control software.

Given the design choice to use compiled FPGA configurations, we recognize that the safety and security drawbacks become a major issue. In the next section we summarize the system security problems that arise and indicate possible countermeasures. We further develop this topic in Chapter 6 starting from the P4 case and then generalize the problem and lessons learned to any other reconfigurable computing systems.

5.3 Safety and Security

In general, executing the code that has originated from a foreign source raises concerns about the security of the system that executes the code. The principles of providing a safe run-time environment for CAML which we use to implement decision modules have been studied[Ale98][LR99]. We focus our attention on providing a safe environment for FPGA implemented modules, which represents a major new issue in all systems augmented with the programmable logic.

The new classes of threat introduced by programmable logic are direct consequence of the ability to modify the device's behavior on the electrical level. Software can only modify the functional behavior of a system and attacks are based by accessing the system's resources beyond a safe limit. The system is secured by either limiting the access to resources or trusting that the program will not violate the system security if the access were allowed. From the hardware perspective, both attacks and regular tasks performed by the software create legitimate cycles on the electrical signal level. In other words, software, by definition, does not have access to resources on the electrical signal level.

Programmable logic, on the other hand, can alter the electrical properties of a system, allowing two completely new classes of attacks: attacks that create logic conflicts and attacks that create invalid bus cycles. We will discuss the general attack classification and protection methods in Chapter 6.

At this point we note that the attacks on the electrical level can be extremely destructive and range from causing unpredictable behavior for the system and temporary malfunctions, to creating permanent physical damage. Since the P4 is a system in which programmable logic devices are programmed with the configurations downloaded from a foreign source, a security threat therefore exists on the electrical level.

Fortunately, efficient methods of attack prevention can be implemented and we describe them in detail in Chapter 6. The most destructive attack which creates logic conflicts can be prevented by statically analyzing the FPGA configuration file and searching for potential logic conflicts. If a potential conflict exists, the configuration is rejected.

The attacks that generate invalid logic cycles can be detected dynamically during the run time or in an off-line live test before using the configuration in the system. To perform a test, it is necessary to model the behavior of the device outputs which is application dependent. We describe the test we have implemented for P4 processing elements in Chapter 6.

5.4 Extension of Signalling Protocol

The signalling protocol described in Section 4.4 and in [Had98] synchronizes the mechanism module insertion and removal assuming that the mechanism implementations are resident on both nodes and identified by the unique ID.

To support the user deployment of new boosters, the signalling protocol must be extended so that the messages can carry the information about the new booster, namely the FPGA device configurations and the CAML object code. While the messages used to synchronize the mechanism insertion and removal are typically short and fit into a single cell, the booster deployment messages carry large amounts of data and a form of adaptation layer must be implemented on top of the ATM layer. Also, a reliable message delivery must be provided through acknowledgements and retransmissions. We have implemented a simple adaptation layer designed for booster deployment which provides segmentation and reassembly assuming the cell format described in Section 4.4. The adaptation layer was designed to provide the minimum necessary functionality and backward compatibility with the P4 systems that do not support boosters deployment.

Booster deployment typically originates from a host, which passes the data to the nearest P4. Once the new booster has been accepted at the P4, it can be propagated to other directly reachable P4 boards and the process continues. Reachability is defined by the existence of a signalling path between two P4 boards. While we assume here that the booster is being deployed from the host to the P4, the same principles apply when propagating the booster from one P4 to the other.

The signalling protocol extension consists of two layers. The “Adaptation Layer” provides for mapping the booster data (FPGA configurations, CAML object code and various control messages) to signalling cells. The "Deployment Layer" is responsible for constructing the booster descriptor at the P4 from the received data and transmitting the requested booster data. The Deployment Layer also provides reliable delivery.

5.4.1 Adaptation Layer

The signalling cell format is shown in Figure 5.3. The first byte is the command code. The second byte identifies the booster to which the command is referring to, and the third byte is the sequence number used during the reassembly to identify lost cells. The fourth byte is the CRC calculated over the first three bytes. The existence of a CRC at the fourth byte provides compatibility with P4 platforms that do not support booster deployment. If such a P4 board receives the deployment signalling cell, it will not recognize the command code and silently drop the cell. A P4 that supports boosters deployment will deliver the cell to the device driver, which will recognize the command code and add the cell to the reassembly buffer.

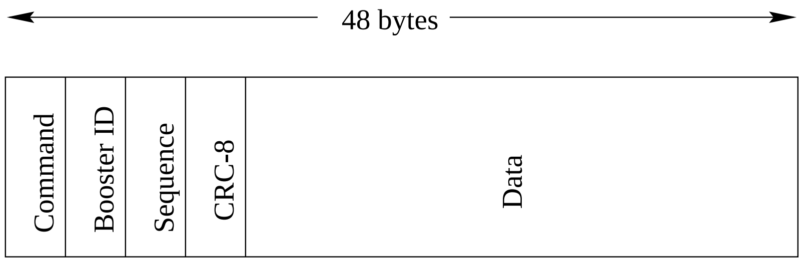

Figure 5.3: Signalling Cell Format

Cells are grouped into packets which carry the higher level information (*e.g.*, FPGA configuration, CAML code etc.). The most significant bit in the sequence number field of the signalling cell is used for packet delineation similarly to the user indication bit in the AAL-5 segmentation and reassembly process[Int93].

The reassembly is implemented in the device driver. Each cell arrival generates an interrupt which transfers the cell to the device driver for reassembly. The first received cell is used to extract the command code, booster ID and the initial sequence number. All subsequent cells must carry the same command code and the booster ID as the first cell. The sequence number must be incremented by 1 modulo 128. If any of the conditions is violated, the device driver aborts the reassembly process and ignores all cells until the end of the packet.

After the packet has been reconstructed, a single `read` system call is necessary to pass the entire packet from the operating systems buffer into the application's address space for further processing. Although generating an interrupt per cell is not a desirable solution for high bit rates, signalling traffic is by the design low bandwidth. Hence, interrupt driven I/O is the appropriate solution. In addition, we control the rate at which the deployment signalling cells are transmitted to conform to the ATM Forum's requirement that the OAM cells may not occupy more than 1% of the total link capacity (Section 3.5 of [ATM94]).

The format of the reassembled packet is shown in Figure 5.4.

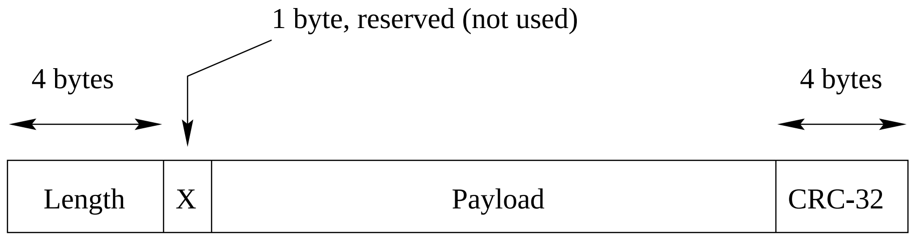

Figure 5.4: Reassembled Deployment Signalling Packet Format

The entire packet is protected from errors with the CRC-32, and the length of the payload is determined by the first four bytes of the packet. If the CRC check succeeds, the reconstructed packet is passed from the kernel space (device driver) to the user space (deployment layer) with the command code and the booster ID extracted from the cell layer.

On transmission, the packet is constructed by adding the length field in front of the payload and calculating the CRC-32. The packet is segmented into 44-byte segments and the command code, the booster ID and the sequence number are added. The P4 calculates the per-cell CRC in hardware. At the host, the CRC must be calculated in software since the appropriate hardware does not necessarily exist. Cells are sent over the signalling channel with the rate control set so that the generated signalling traffic does not exceed 1% of the total available bandwidth.

5.4.2 Deployment Layer

The deployment layer is responsible for constructing deployment messages, interpreting received messages and constructing a booster descriptor. It also provides reliable message delivery. The semantics of a packet depends on the associated command code.

Deployment is initiated by the host[3] which sends the `DEPLOY_REQUEST` command to the P4. The booster is always deployed in a booster-debooster pair and the request carries a list of MD-5[Riv92] hashes of the following modules:

1. The FPGA configurations of the booster's mechanism module (configurations for up to four devices).
2. The FPGA configuration of the booster's monitor module.
3. The CAML object code of the booster's decision module used when the mechanism is not engaged.
4. The CAML object code of the booster's decision module used when the mechanism is engaged.
5. All of the above modules associated with the debooster.

A combined list of MD-5 hashes uniquely identifies the booster and the associated debooster, while individual MD-5 hashes uniquely identify modules that constitute a booster. After receiving the requested packet, the P4 will check existing booster descriptors and search for matches in the received MD-5 list. A match indicates that the P4 already has the module and that deployment is not necessary. The P4 responds with the `DEPLOY_RESPONSE` which carries the list of locally assigned 8-bit ID's for each module needed. Three cases may occur:

1. The entire list of received MD-5 hashes matches the entire list of MD-5 hashes in the existing booster descriptor

[3] Note that we discuss deploying the booster from the host to the nearest P4 and use words "host" and "P4" to refer to the entities in the process. The same principles apply for propagating the booster from one P4 to another

2. All received MD-5 hashes match the existing MD-5 hashes, but the booster descriptor containing the entire list of received MD-5 hashes does not exist.

3. Some or none of received MD-5 hashes match MD-5 hashes of existing modules.

The first case indicates that the booster and the debooster already exist at the P4 and no new module is needed. No new descriptors will be created at the P4 and the response will indicate the end of deployment procedure.

In the second case, all the necessary modules are present at the P4, probably constituting other boosters, but the booster and debooster consisting of the requested combination of modules does not exist. The P4 will create new booster descriptors, assign the IDs, and respond to the host by ending the deployment procedure.

Finally, the third and most common case indicates that some or all modules are missing and need to be deployed by the host. The P4 will assign local IDs to all missing modules and request them from the host by sending the `DEPLOY_RESPONSE` message with the list of locally assigned IDs. Two new booster descriptors will be created but the booster will not be activated until all the necessary modules are received.

Multiple deploy requests may be pending simultaneously. For each new request the host starts the timer and retransmits the request on each timeout. When the response is received, the timer is stopped. After the maximum number of attempts has been exhausted, the host gives up requesting deployment, assuming that the targeted P4 is either not alive or it does not support booster deployment.

The received `DEPLOY_RESPONSE` message is analyzed by the host and for each requested module an FPGA configuration or a CAML object code is sent. Each module is sent in a separate packet and must be acknowledged independently. The host creates the list of modules sent and starts the timer for each of them. When the acknowledgement is received, the module is removed from the list and the timer is stopped. On every timeout the module is retransmitted and the timer is restarted.

For each received module, P4 fills in the blanks in the booster descriptor table and sends an acknowledgement. When the descriptor is completed, CAML decision modules are dynamically linked into the run-time system and the control thread associated with the booster is created. The control thread will start executing the signalling protocol described in Section 4.4 and invoking decision modules.

5.5 The Complete Software Architecture

We now describe the complete software architecture of the system which supports booster deployment. Figure 5.5 shows the extension of the architecture described in Section 4.8. For simplicity, we have omitted all data and control paths except those discussed in this section.

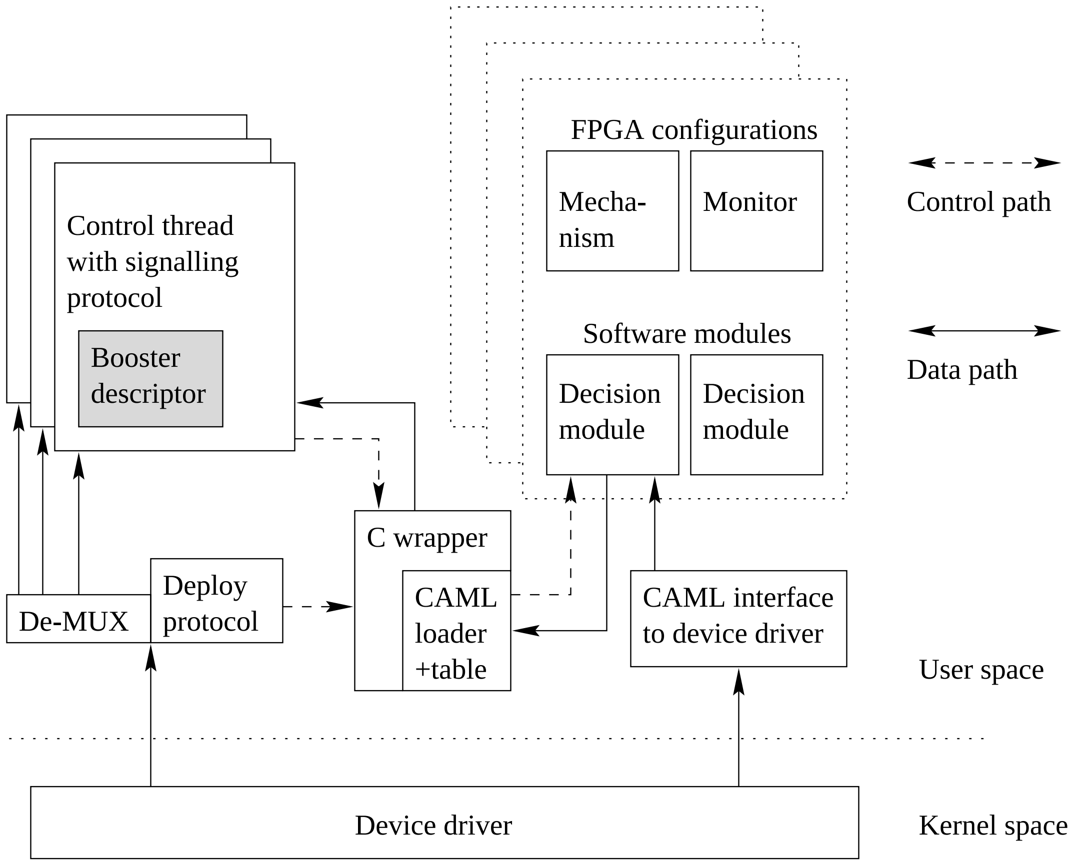

Figure 5.5: P4 software architecture with support for user programmability

The main thread consists of a signalling demultiplexer (De-MUX) and an extension of the signalling protocol for boosters deployment. An arriving deployment packet is demultiplexed from other signalling cells and processed by the protocol as described in

Section 5.4.2.

After the booster descriptor is constructed, the CAML implemented decision module is loaded and dynamically linked using a "Dynlink" module[Ler99]. The loaded decision module links with the control program using the hash table, similar to the approach taken in loading services in PLAN implementation[Hic98]. The unique decision module name generated by the system is passed to the loader. The module is loaded and its top level function is evaluated resulting in a module registration. The hash table entry associates the module name with the entry point.

To invoke a decision module, the control thread reads the module name from the booster descriptor and makes a "callback"[Ler99] into the CAML code that implements the hash table lookup. The hash table returns an entry point to the CAML decision module implementation and invokes the module. The decision module reads the information from the monitor via the CAML device driver interface, which is essentially a C-implemented `ioctl` system call wrapped into the CAML function. The access to the device driver is limited to reading from the monitor, so that it does not jeopardize the safety of the system. For improved safety, it would be straightforward to limit the number of reads from the device driver and the decision module execution time to prevent denial of service attacks if the decision module never returns. Processed monitor data is converted into a decision and returned to the control thread where the policy decision is enforced.

5.6 Deployment Latency Measurements

Previous sections provided the qualitative description of our programmable network infrastructure. We now present the experimental results for deployment latency. We define the deployment latency as the time from when a host initiates the deployment (*i.e.*, sends `DEPLOY_REQUEST`) until the new booster becomes available at the targeted P4 node (*i.e.*, policy thread is created).

We measured the deployment latency assuming that the host is deploying the FEC booster-debooster pair described in the Chapter 4. The deployed booster consists of three FPGA configuration (encoder, decoder and BER monitor) and two CAML-implemented decision modules. We provide a breakdown of deployment latency on both the host and the P4 side and discuss which portions of the latency can be reduced and how.

5.6.1 Host Side

On the host side, the deployment procedure is implemented with two threads. The main thread implements the complete receive path and the deployment layer of the transmit path. The adaptation layer is a separate thread. We will refer to this thread as "the TX-thread". The main thread and the TX-thread communicate using a message passing mechanism. When the main thread has a message to send, it enqueues it in the transmit queue and signals the TX-thread that the message is available. The TX-thread wakes up and transmits all messages in the queue enforcing the nominal gap (in our experiments set to $300\mu s$) between the consecutive cells so that the bandwidth demands for the signalling traffic are kept low.

Figure 5.6 shows the timing diagram of the booster deployment procedure on the host side. The deployment phases and their respective execution times are:

1. The deployment layer constructs the request by calculating the MD-5 hash of each configuration and the decision module that constitute the new booster (latency of this phase is T_{ldr}).

2. The adaptation layer calculates the CRC-32 of the deployment message, enqueues the message and signals the TX-thread that the data are available for transmission (latency: $T_{enqueue}$).

3. The TX-thread wakes up and dequeues the message (latency: $T_{dequeue}$).

4. The TX-thread sends the message (latency: $T_{transmit}$).

5. The host waits for the P4's response (latency: T_{wait}).

6. The adaptation layer receives and reassembles the response and checks the CRC-32 (latency: $T_{receive}$).

7. The deployment layer processes the response and constructs the messages (latency: $T_{processing}$).

8. The adaptation layer calculates the CRC-32, enqueues the messages and signals the TX-thread (latency: $T_{enqueue}$).

9. The TX-thread transmits the messages (latency: $\sum T_{transmit}$).

10. The host waits for the last acknowledgement (latency: T_{wait}).

11. The host receives and processes the last acknowledgement (latency: $T_{receive} + T_{ack}$).

A decoupled transmission from the higher layer enables the system to take advantage of the parallelism and reduce the deployment latency. Since the transmission time ($T_{transmit}$) is typically greater than the time to construct and enqueue the message, all deployment messages that constitute the new booster will be in the queue before the TX-thread has finished the transmission. Hence, the $T_{enqueue}$ will not count multiple times towards the deployment latency. Similarly, all acknowledgements will be processed before the transmission of the last message is finished and the time spent while waiting for the P4 to acknowledge the message will count only once.

Table 5.1 shows the 90% confidence intervals for each portion of the deployment latency that is due to the host. The times that depend on the message size are given for three cases: 113-byte `DEPLOY_REQUEST` message, 600-byte decision module (CAML object code) and near 64-Kbyte configuration data. The `DEPLOY_REQUEST` message construction time (which includes the MD-5 calculation) is given for the FEC booster consisting of three configurations and two decision modules. Measurements have been done on the 300MHz PC with Pentium II running Linux kernel 2.0.29.

The most costly operation is data transmission which is the longest for the FPGA configuration. The primary reason for the long transmission time is the requirement to keep low bandwidth demands of the signalling traffic enforced by the TX-thread. The transmission time can be reduced by relaxing the limitation on signalling bandwidth (which would most likely require using DMA instead of the interrupt driven I/O).

Following this is the response processing time, which consists of constructing the deployment message for each module (FPGA configuration or CAML object code) and creating the auxiliary data structures necessary for reliable delivery (list entries and timers). A portion of this time is due to the operating system (*e.g.*, system calls related to timers and

		90% confidence intervals [μs]
T_{ldr}		[8287.17, 8347.94]
$T_{enqueue}$	request	[90.02, 90.97]
	configuration	[1647.36, 1699.97]
	decision mod.	[24.17, 25.49]
$T_{dequeue}$		[211.33, 231.67]
$T_{transmit}$	request	[3665.08, 3957.36]
	configuration	[868181.40, 874991.50]
	decision mod.	[8881.76, 8913.90]
$T_{receive}$		[14.46, 20.87]
$T_{processing}$		[14381.65, 14783.01]
T_{ack}	last	[148.36, 166.30]
	other	[9.54, 12.91]

Table 5.1: Confidence intervals (90%) for deployment latencies at the host side

memory management), while the other portions are defined by the size of the deployed booster.

The third most costly operation is constructing the `DEPLOY_REQUEST` message primarily due to the MD-5 calculation. This portion of the latency can be improved by using the faster processor.

Calculating the CRC-32 for each deployment message takes up a significant time for very long messages such as the FPGA configuration. Due to a parallelism between the main thread and the TX-thread, some of this cost can be hidden as the CRC-32 for the next packet can be calculated while the current packet is being transmitted as shown in the Figure 5.6 (note that all deployment messages are enqueued while the first is still being transmitted). Since the time to calculate the CRC-32 grows linearly with the length of the packet, the overall deployment latency be reduced if shorter packets (*e.g.*, decision modules) were transmitted first.

Acknowledgement processing is a short operation which contributes very little to the overall deployment latency. However, it can be noticed that the last acknowledgement takes much longer time to process than other acknowledgements. The additional time is due to the operating system and is unfortunately irreducible, as all auxiliary data structures must be cleared when the deployment has been successfully completed.

5.6.2 P4 Side

On the P4 side, the device driver receives signalling cells using the interrupt driven I/O. The message is reassembled in the device driver and the user process is signalled when the message is ready. The user process waits on `select` and proceeds when the new message becomes available. The incoming message is copied from the kernel space and the CRC-32 is checked. If the message passes the check the deployment layer processes the message and sends the response. The response is sent by the TX-thread using the mechanism described in the Section 5.6.1.

Figure 5.7 breaks down the deployment procedure into phases for the P4 side. The phases and their respective times of execution are:

1. The main thread waits on `select` for the request to arrive (latency: T_{wait}).

2. The adaptation layer copies the message from the kernel space into the user space and checks the CRC-32 (latency: $T_{receive}$).

3. The deployment layer checks which modules already exist at the node, assigns the booster ID and constructs the response (latency: $T_{request}$).

4. The adaptation layer calculates the CRC-32 of the response, enqueues the message and signals the TX-thread that the data are available for transmission (latency: $T_{enqueue}$).

5. The TX-thread wakes up and dequeues the message (latency: $T_{dequeue}$).

6. The TX-thread sends the message (latency: $T_{transmit}$).

7. For each deployment message (FPGA configuration or decision module) the following steps are repeated:

   (a) The message is copied to user space and the CRC is checked ($T_{receive}$).

   (b) The module is made node resident, and the entry in the booster descriptor table is updated (T_{update}). This step could include checking if the received FPGA configuration is safe as described in Chapter 6.

		90% confidence intervals [μs]
$T_{receive}$	request	$[1161.97, 1167.69]$
	configuration	$[10200.98, 11110.68]$
	decision mod.	$[1090.99, 1320.16]$
$T_{enqueue}$		$[21.20, 33.64]$
$T_{dequeue}$		$[164.05, 165.95]$
$T_{response}$		$[937.50, 965.50]$
$T_{transmit}$		$[515.15, 609.19]$
T_{ack}	configuration	$[16.03, 18.30]$
	decision mod.	$[180.00, 238.00]$
T_{update}	configuration	$[10549.32, 11593.15]$
	decision mod.	$[2540.42, 2882.38]$

Table 5.2: Confidence intervals (90%) for deployment latencies at the P4 side

(c) The acknowledgement is constructed and sent by the TX-thread ($T_{ack}+T_{enqueue}+T_{dequeue}+T_{transmit}$).

Table 5.2 shows the 90% confidence intervals for each portion of the deployment latency that is due to the P4. The measurements have been done on 166MHz Pentium running Linux 2.0.34. The results do not include any security checks (These are discussed in the next chapter). The time to copy the incoming message to the user space and check the CRC-32 is given for the same three cases as in the previous section. The transmission delay is not dominant in this case; all messages transmitted back to the host are acknowledgements and responses which are typically short.

The dominant performance cost at the P4 side is updating the booster descriptor table and copying the received data from kernel space to user space. Updating the booster descriptor table includes calculating the MD-5 hash of the received data, searching through the table of known modules and copying the data from the receive buffer to the entry in the table. Copying the data from kernel to user space consists of the system call and the actual data transfer. Both times can be reduced by eliminating data copying operations (*i.e.*, implementing zero-copy protocol stack). Since in a production system, the P4 controller would be an embedded system, the heavy-weight system calls could be eliminated.

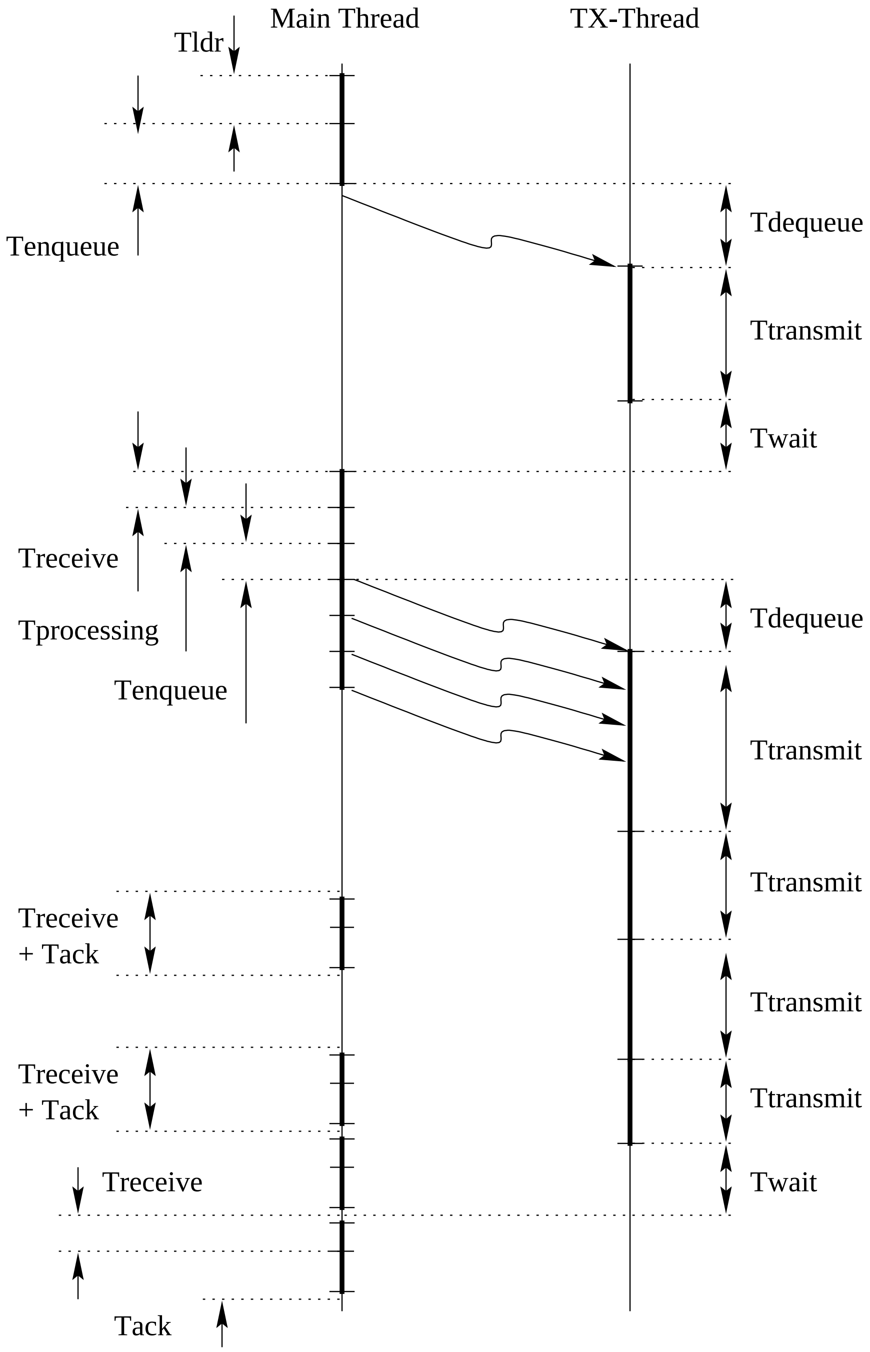

Figure 5.6: The booster deployment procedure on the host side

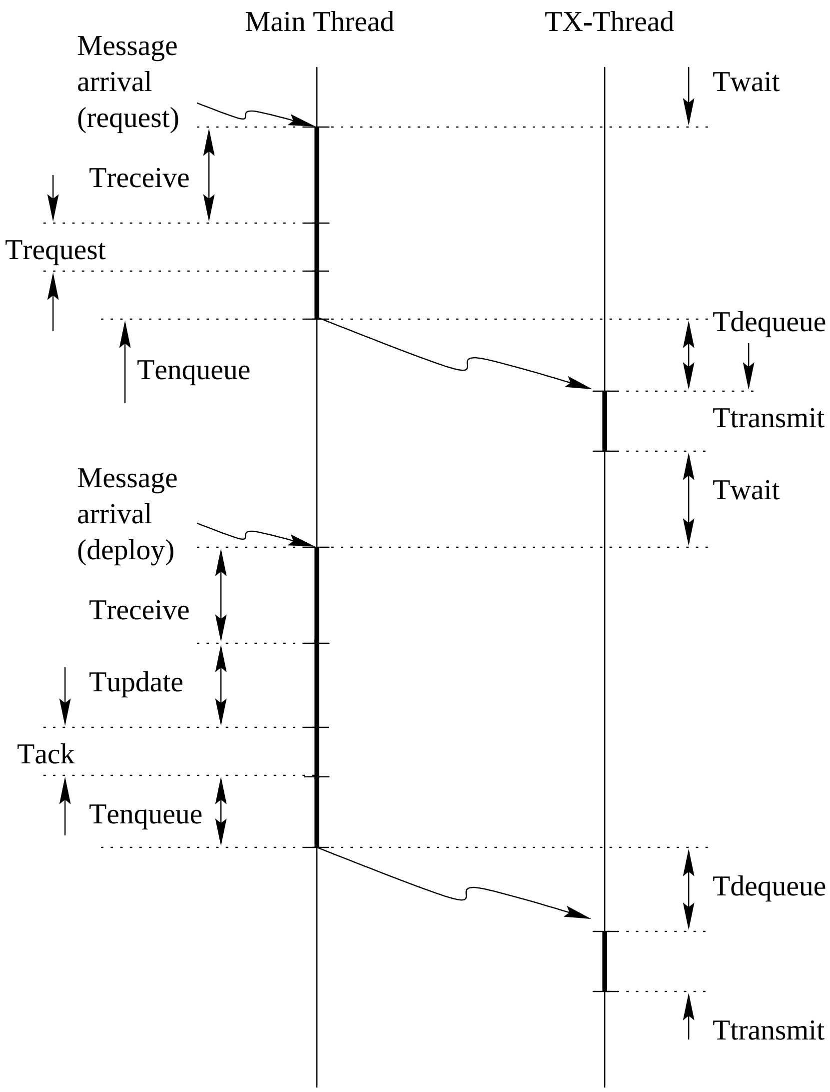

Figure 5.7: The booster deployment procedure on the P4 side

Chapter 6

Reconfigurable Hardware Security

Our new approach to network programmability raises novel security concerns. More specifically, we are downloading modules implemented in hardware that originated from a foreign source. Intuitively, such a run-time environment could allow the implementation of hardware equivalents of computer viruses that attack the system on much lower level, namely, the electrical signals level. As we indicated in Section 5.3, an attack on the electrical signal level could potentially cause physical damage to the system, which represents a completely new class of threat.

In this chapter we demonstrate that the threat is realistic. Electrical level attacks are indeed possible and, they can achieve the anticipated result: physical destruction of the hardware.

We also discuss the possible methods of protection from such attacks. An obvious method, common to software and FPGA-based systems, is establishing a trust relationship between the entities that distribute and use the code or FPGA configuration. The same algorithms for encryption and authentication that apply to software systems also apply to reconfigurable computing systems. Cryptographic methods have been studied in work by others, and are not examined further here.

As an alternative to cryptographic methods, software systems can deal with the safety and security issues through the use of type-safe programming languages the proof-carrying code[Nec97]. However, these methods are not directly applicable in reconfigurable computing. Our protection methods for FPGA-based systems also represent an alternative to the cryptographic methods and are reconfigurable computing specific.

We examine the features of programmable logic that a malicious user can exploit to achieve his objective and discuss the methods of preventing and detecting an attack.

The existence of a threat is a direct consequence of adding the programmable logic to the system and it is not limited to P4. Every system that opens programmable logic to the user is susceptible to attacks we are describing. Hence, the results of this chapter are general and apply to any reconfigurable computing system. We anticipate that with wider adoption of reconfigurable hardware, future systems will combine a general purpose CPU with a programmable logic accelerator to achieve high performance. In such systems, reconfigurable hardware security will be a foundation for providing a safe and secure run-time environment.

No work by other researchers we are aware of has addressed the security in the context of protecting the FPGA devices from downloading a malicious configurations. We have recently published preliminary results on this topic[HUS99] which is the first discussion on FPGA security. In this chapter we extend that discussion.

Besides describing the security of P4, a much more ambitious objective is to open a new research direction in the reconfigurable computing field. Once reconfigurable computing goes mainstream, the new threats to system security will have to be thoroughly understood. In that sense, our results are a general analysis of reconfigurable computing system, not limited to the P4. We focus on the P4 platform only when we discuss a specific application or address platform dependent issues.

6.1 Types of Attack

To classify the wide range of potential attacks to a system, we have created three categories based on the type of threat to a reconfigurable computing system:

1. Electrical Signals Level: The attacker creates electrical conflicts either inside the device or at pins connecting the attacked device to other components of the system. The goal of the attack is to physically destroy system components. We call this class of threat a Malicious Electrical Level Threat (*MELT*).

2. Logic Signals Level: The attacker generates signals which are electrically correct, but logically make no sense to other devices. For example, a processing element on P4 (see Chapter 3) may generate invalid write cycles to the FIFO buffer which could result in missing or bogus data causing a loss in cell integrity, which further results in unpredictable behavior of other processing elements. We call this class of threat a Signal Alteration Logic Threat (*SALT*).

3. Software-like Attacks: Finally, the attacker may generate legitimate cycles which together compose an execution of a malicious task. This attack level is equivalent to the attacks in software systems. We call this class of threat a Higher Abstraction Level Threat (*HALT*).

MELT represents an interesting, and most destructive, form of attack enabled by the addition of reconfigurable hardware. SALT attacks may cause unpredictable behavior in the system, but cannot directly cause physical damage. They are less destructive than MELT, but their detection and prevention can be very difficult, mainly because any such prevention requires a rather complete model of the system in which the device is embedded. Finally, HALT attacks should be treated as equivalent to malicious software code and are thus not FPGA-specific. They are harder to detect since the device has no model for the valid behavior of systems scaffolded on top of it. In such instances, a defense should be based on establishing a trust relationship between the source of the configuration and the system executing it (*i.e.*, when configuring the FPGA device). A HALT attack is thus a classic security problem and therefore neither novel nor of particular interest for our work.

We focus our attention on MELT and SALT attacks. For the MELT class we consider two categories based on where the logic conflict is created: internally or externally. For both categories, we provide a general analysis. The only assumption we make is that a foreign (untrusted) source provided a configuration for the FPGA device.

For the SALT class of attacks, we provide a case study. Because of the inherent application and platform dependency, we analyze this attack class in the context of the P4 and try to draw some general conclusions.

6.1.1 Categories of MELT Attack

The attacker's goal at the electrical level is to physically damage the system. To destroy a system component, the attacker must create high currents either inside the device or at its input/output pins. The latter case can be easily realized provided that the attacker is familiar with the board level architecture of the system. It is necessary to know which pins of the attacked device are supposed to be configured as inputs (*i.e.* connected to outputs of external devices) and configure them as outputs.

This will result in a potential conflict in logic levels creating a high current through the output transistors of both the attacked device and, for example, another device connected to it, as shown in Figure 6.1.

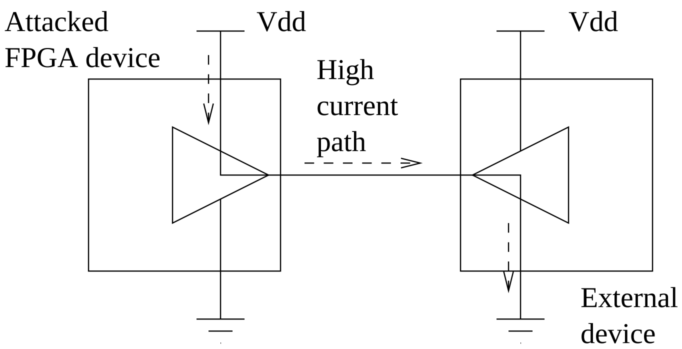

Figure 6.1: Electrical conflict at I/O pins

Either device may be destroyed if a high current is applied for a sufficiently long period of time. In addition to knowing the board level architecture, the attacker must have some insight on the behavior of external signals, as high currents exist only if the attacked device outputs the logic complement of signals applied to it. Since the compiler is not aware of the board level system architecture, such a malicious configuration represents a legitimate design from the device's perspective and a defense using compiler techniques does not appear possible.

The second group of electrical level attacks attempts to create high currents inside the device by programming it with a configuration that creates a logic conflict in the internal connections. Most FPGA devices use pass gates to connect logic blocks to routing resources. This represents an opportunity for creating an internal conflict if two (or more) logic blocks are configured to drive the same routing resource, as shown in Figure 6.2.

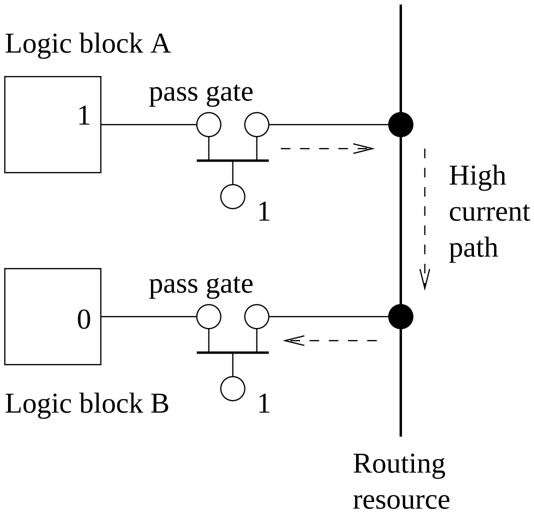

Figure 6.2: Electrical conflict at I/O pins

In contrast with the previous example, such a configuration cannot be generated by a valid compiler. However, it is in fact possible to modify the compiler output file (*i.e.*, the device configuration data) and create internal logic conflicts.

In the next section we demonstrate the construction of such a configuration, using the Altera EPF8636ALC84-4 device[Alt98c] as an experimental platform (any vendor's devices will exhibit similar properties), using no Altera proprietary information, and rendering the device inoperable.

The board we used for experimenting with electrical level attacks is the old version of the P4[HS97] which used devices from the Altera FLEX8000[Alt98c] family. Because of the destructive nature of the experiments, we used the obsolete version of the board rather than the one needed for other experiments. Due to similarities between the FLEX8000 and FLEX10K architectures, results obtained on one family directly apply to the other. In fact, since the primary source of vulnerability is a connection of multiple pass gates to the same routing resource, a technique used by all major FPGA families, our results can be easily reproduced on any FPGA device.

6.2 An Attack Example

A connection of multiple logic blocks to the same routing resource via a pass gate represents a vulnerable point in FPGA devices. In the Altera Flex 8000 family, the logic elements (LE) from logic array blocks (LAB) in the same column share a column interconnect.

For example LE(1) in LAB(A1) in Figure 6.3 will share the column interconnect with LE(1) in LAB(B1). Therefore, if we could program the LE(1) in LAB(A1) and LAB(B1) to output complementary signals and connect to the shared column interconnect, an internal conflict would be created. This conflict pattern can be replicated as many times as the device size allows, increasing the device power consumption up to a level sufficiently high to overheat and destroy it.

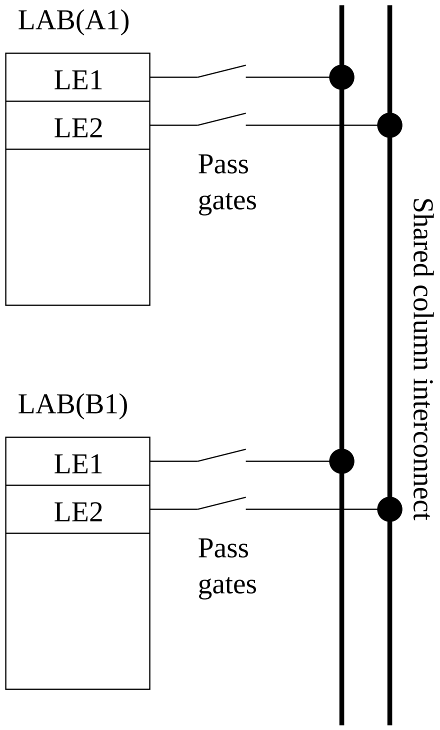

Figure 6.3: Shared column interconnect in Altera Flex 8000/10K devices

To construct an internal conflict, we analyzed the configuration files of simple logic designs and compared the differences resulting from changing the logic element assignments and the logic functions. We identified locations in the configuration file which correspond to the logic element configuration and connections to column interconnect. To create a conflict we used two designs, one that utilizes a 4-input NOR gate in row A of the device and one that utilizes a 4-input OR gate in row B[1]. We created a configuration with internal conflicts by mixing the logic element configurations and column interconnects from two previously described configurations. Since the configuration files for these devices do not use a global checksum or CRC, this cut-and-paste type of attack can be easily realized. Even if a global checksum or CRC existed, the device would not be secure since the attack would require only slightly more effort.

We have experimentally verified our claim by downloading configurations with internal conflicts into the device and measuring the supply current. No clock was applied so the measured results represent the quiescent supply current which is typically very low. The results of our measurements are shown in Figure 6.4. With only one conflict the quiescent current is greater than the maximum of $10mA$ specified by the datasheet[Alt98c] and it grows almost linearly with the number of conflicts. A small non-linearity appears due to the fact that the mobility of carriers in silicon drops with the rising temperature, causing some current flow to be reduced. Typically, the supply current will have an overshoot after the device is configured and will fall as the device heats up until it reaches steady state. Despite this negative temperature feedback, quiescent current can grow arbitrarily and the upper limit is determined only by device size – that is, the number of possible logic conflicts.

Although it is not possible to *guarantee* the physical destruction of a device, the attacker would normally try to make a device operate in the unsafe region and increase the *probability* of its destruction. As an illustration we will determine the critical current at which the junction temperature exceeds the maximum of $T_J = 135^oC$ specified by the datasheet[Alt98c]. We will show that this temperature is easily achieved with a relatively modest number of logic conflicts.

The thermal resistance of the PLCC84 package is $\theta_{JA} = 35^oC/W$[Alt98a] and assuming the ambient temperature of $T_A = 25^oC$, we can easily calculate the maximum power consumption:

[1]The logic function used is arbitrary as long as the functions in row A always output the complement of functions in row B and are sufficiently "complex" so as to prevent the compiler from placing the logic in I/O blocks.

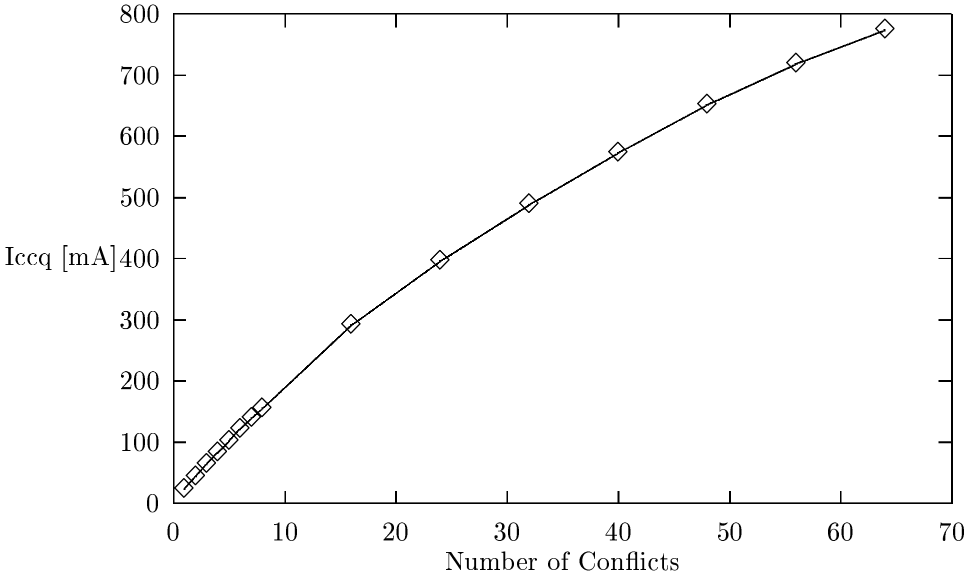

Figure 6.4: Quiescent current as a function of number of logic conflicts

$$P = \frac{T_J - T_A}{\theta_{JA}} = \frac{135 - 25}{35} = 3.143[W] \tag{6.1}$$

For the power supply voltage of $V_{dd} = 5V$, we calculate that the maximum allowed supply current is $I_{cc}^{max} = 629mA$. From the graph in Figure 6.4, it is clear that this limit can be easily exceeded with the quiescent current alone (*i.e.*, without applying the clock) if the number of logic conflicts is greater than 50. During our experiments, we have observed extensive device heating. In several trials, this resulted in physical destruction of the device, manifested through the device's inability for further reconfiguration after being exposed to the high supply current.

In addition to being a threat to the attacked device, extensive heating could also increase the ambient temperature violating the temperature design assumptions of other systems. For systems designed to be physically small and densely packed, temperature may be a critical issue and increasing the ambient temperature could make the entire system operate in an unsafe region, leading to larger scale failures.

6.3 Countermeasures

The design space for an attacker in systems with run-time reconfigurable hardware contains the design space of the CPU based systems.

Our goal is to study the defense methods only for attacks specific to reconfigurable hardware and to provide a safe environment on the electrical and logic signal levels (*i.e.*, protect from MELT and SALT attack types) so that the attacker's design space is reduced back to its software subset.

6.3.1 MELT Attack Prevention and Detection

A necessary condition for a logic conflict is that two or more logic blocks are connected to the same routing resource or that an output pin is connected to the output of an external device. If the first condition is met, an internal logic conflict *may* exist, while the second condition allows logic conflicts at the I/O pins. It is important to note that these are only the *necessary* conditions, which implies that ensuring that the necessary conditions are not met would ensure the non-existence of logic conflicts. Therefore, the attack can be prevented by not allowing the necessary condition to be satisfied.

A sufficient condition for a logic conflict is that internal logic blocks or I/O blocks drive the same routing resource with complementing logic levels.

Satisfying the necessary condition is a property that a configuration may or may not have before downloading: it is an "off-line" property created by the attacker at "compile"[2] time. The sufficient condition can only be satisfied at run-time, after the configuration has been downloaded into the device.

6.3.1.1 Attack Prevention

We define two classes of defense from MELT attacks: attack prevention and attack detection. Attack prevention methods ensure that none of the necessary conditions is satisfied. Attack detection allows the existence of potential logic conflicts and relies on high-current detection at run-time.

[2]Here we assume that the "compilation" may include using a valid compiler and manually modifying the resulting configuration, or using a maliciously modified compiler.

Internal logic conflicts can be made technically impossible by avoiding the use of pass gates in the device architecture. Although it would completely eliminate the opportunity for an electrical level attack on the internal logic, this approach is not feasible for economic reasons. For example, a set of pass gates driving the same routing resource can be replaced with a multiplexer. In VLSI circuits, a multiplexer is a set of pass gates with additional logic (*e.g.*, N-to-2^N decoder) which does not allow two gates to drive the output at the same time[WE93]. Clearly, the additional logic occupies space on a silicon die, reducing the amount of logic available to the user. It is likely that the device manufacturers will tend to exclude all "not-deemed-necessary" circuitry as device density increases.

As a second example, the device architecture may be designed without shared routing resources. The Altera FLEX8000 architecture[Alt98c] has these properties for row interconnect lines. However, column interconnect lines are shared in this architecture making it still vulnerable to MELT attacks. As in the first example, it is likely that high density devices will tend to share routing resources among logic blocks. This trend can be clearly seen from the evolution of Altera FLEX families where the older and low density FLEX8000 family had only column interconnects shared among the logic blocks, while the newer FLEX10K[Alt98d] family allows logic blocks to share row interconnects as well.

Even if a device that makes internal logic conflicts impossible existed, potential logic conflicts on the I/O blocks would still not be eliminated. Since the I/O pins may be configured as either outputs or inputs, a malicious user could still create logic conflicts by configuring the pins to conflict with external circuitry.

As a method of attack *prevention*, we propose pre-downloading analysis of the configuration file. The configuration file is a bit map containing the configuration of logic blocks (states of lookup tables and internal connections) and connections to routing resources. Prior to downloading the configuration, the system can search the file for multiple logic blocks driving the same routing resource. Each such connection is a potential logic conflict and cannot represent a meaningful design. Hence, the device configuration with such conflicts should be rejected. The configuration file also carries the I/O block configuration data, so the system may check if the output pins will be configured as expected, eliminating potential external conflicts (*i.e.*, not allowing output-to-output connection). Since the system is, or can be made aware of, the board level architecture, the off-line checking of the I/O blocks against external conflicts is possible. At a very high level, the pre-downloading analysis is analogous to type-checking in strongly typed programming languages.

The advantage of attack prevention through analysis is that the attack can be prevented before the device is exposed to a malicious configuration. The major disadvantage is in the time necessary to perform the analysis, which adds to the download time. Users would also need to know the device configuration file format, something which device vendors are reluctant to make public[All97].

Configuration file analysis eliminates internal logic conflicts. External conflicts on unidirectional pins are also eliminated. The problem of potential logic conflicts still exists on bidirectional lines and open-drain outputs. Wiring these lines together still composes a legal circuit, but the logic conflict may be created during run-time. For example two bidirectional lines may choose to drive the bus with complementary logic levels at the same time, or two wired open-drain lines may go out of the high impedance state at the same time driving the line with complementary logic levels. This situation can occur only during run-time and it is almost impossible to predict it with static analysis.

The MELT attacks through bidirectional and open-drain pins can be prevented with a safe board level design. If the bus driver were implemented as an external circuitry, the FPGA device will no longer need to use the bidirectional lines and the configuration analysis would be able to simply declare unsafe and reject all configurations with bidirectional lines regardless of where they are connected to. The identical strategy can be applied to open-drain outputs.

Figure 6.5 shows an unsafe board level design. The data bus is composed by directly wiring bidirectional lines of the two FPGA devices. If the functionality of both FPGA devices were correct, this would not represent a threat since the two devices would never drive the bus at the same time, assuming that a form of arbitration logic exists. In reconfigurable systems, the FPGA devices are typically downloaded with a foreign configuration and correctness can not be assumed. Two devices may either maliciously of incidentally drive the bus at the same time (not respecting the arbitration logic) and cause the logic conflict at the bus.

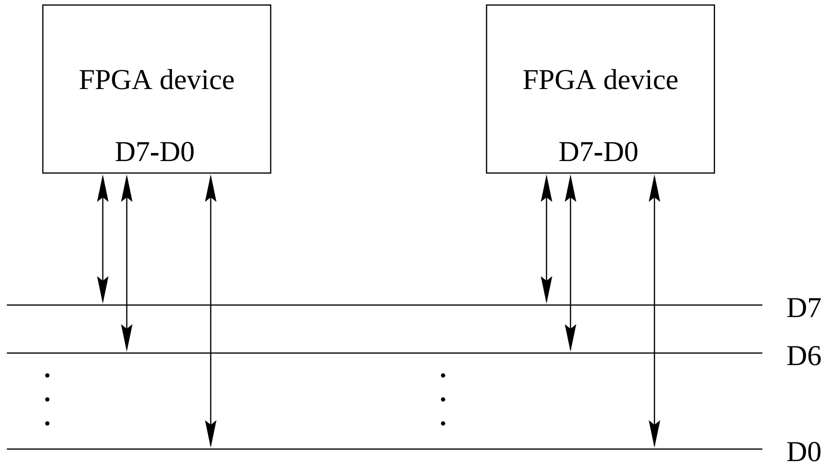

Figure 6.5: An unsafe board level design

An example of a safe board level design is shown in Figure 6.6. The pins on the FPGA devices are now unidirectional and the system can require this property from all FPGA configurations. Note the addition of a new control output which the device uses to request an access to the bus and the priority encoder. If all devices behave in a disciplined manner, only one device at a time will make a request to drive the bus and the priority encoder will behave as if it were not there: it will simply pass the request signal to the bus drivers and the device will gain the access to the bus. In the case of conflicting requests, the priority encoder will pass only one input signal to the bus driver and force others to be inactive. As a result, only one device will be able to drive the bus. Additionally, the priority encoder may detect simultaneous bus requests and generate an alarm, alerting the system to suspicious activity.

The priority encoder is a hard wired circuit and can be assumed correct if the board level design is correct. Obviously, the price of safety is in the need for external "glue-logic" and the increased pin counts. Since most modern FPGA devices have very large pin count (several hundreds of user pins) the addition of a few new pins should not represent a major problem. The external circuitry will make the bus response slower which could affect the overall system performance, which is generally a price of safety in all systems.

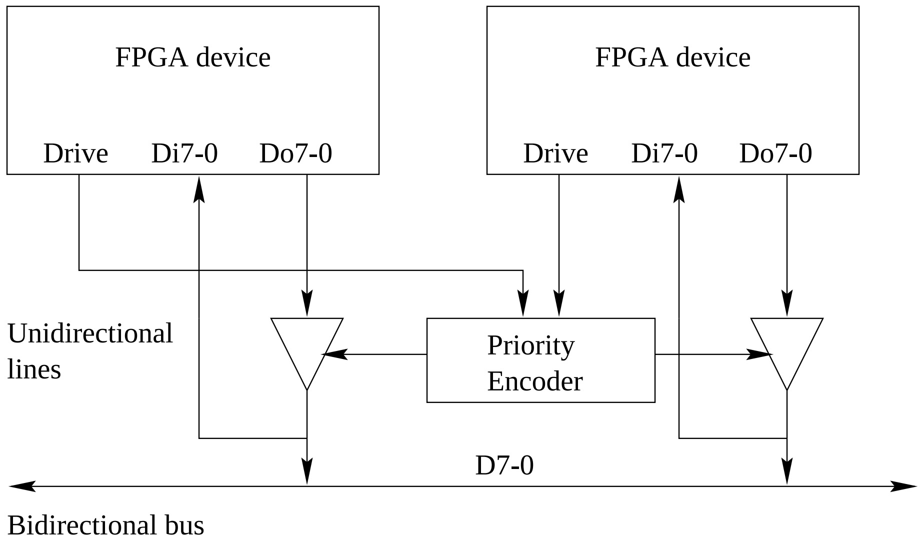

Figure 6.6: A safe board level design

6.3.1.2 Attack Detection

Although configuration file analysis is an efficient method of attack prevention, sometimes this option is not viable or the processing overhead may be considered too high. A typical example would be in applications which reconfigure the FPGA device several times during the task execution so that the download time becomes a performance critical factor. In addition, the configuration file format may be an inhibiting factor in preventing the attack through off-line analysis. In such cases the run-time attack detection may be used.

Electrical level attacks have the property that they must generate a high current either internally or at the I/O pins. In the case of internal logic conflicts the high current drawn causes a high current at the power supply lines. If the logic conflict is generated by the combinational logic, as in the example from Section 6.2, a high current will be present in quiescent mode, and it will be typically greater than the maximum allowed quiescent current[3].

The attack can be detected by measuring the supply current after the device has been configured, but before applying the clock. Power can then be immediately disconnected from the device if the maximum current is exceeded. This detection method has neither

[3]We have experimentally verified this claim only for Altera Flex8000 devices, but we believe that other device families also have this property.

of the disadvantages of the analysis method. The key disadvantage is in the need for additional protection circuitry, and the existence of a short current pulse before the attack is detected.

Another major disadvantage of this method is that it is hard to guarantee the detection of logic conflicts generated by sequential logic. Consider the case where the outputs of two sequential circuits are connected to the same routing resource, but after the initialization they output the same logic level. The logic conflict does not exist because the outputs are not in the conflicting state. If at some point in time the two circuits start driving the complementary logic levels, a conflict will appear and the high current will exist in the attacked device. However, this can only happen after the clock has been applied since the sequential circuit cannot change the state, nor its output, otherwise. The circuit will no longer be in the quiescent mode and the total current drawn from the power supply will typically be much higher than the quiescent current. This may further result in not detecting the attack or in bogus attack detection.

6.3.2 SALT Attack Detection and Prevention

This attack class is far less destructive than MELT attacks. The goal of the attacker is not to destroy the device, but to cause unpredictable behavior of the system by generating invalid cycles. To reliably detect the SALT attack, the system needs a rather complete model of a "valid cycle" and a set of test vectors that would fully cover the downloaded circuit functionality. These properties make the the SALT attack detection difficult and highly application dependent.

If the configuration has been checked for MELT attacks (*e.g.*, by static analysis), it can be safely downloaded into a device isolated from the rest of the system. The SALT attack in an isolated device cannot impact the system. From this observation, it follows that the SALT attacks can be screened by first downloading the received configuration into the test-device whose inputs are driven by the test vector generator and outputs scanned by the data acquisition module. If the test vector generator can cover complete functionality of the device and if the data acquisition module has a complete model of valid bus cycles, the device can be checked for SALT attacks.

Once the safety has been verified for both attack classes, the configuration can be safely downloaded into the regular processing path. The questions that remain open are how to determine test vectors and how to compare the scanned data with valid bus cycles. The answer depends on the application and in this section we provide a case study of the FIFO buffer interface used on the P4 processing elements.

Figure 6.7 shows the FIFO buffer interface of the P4 processing element. The processing element reads from its FIFO buffer and writes into the buffer associated with the next device. The FIFO buffer we use in the P4 is the Cypress CY7C4251[Cyp95] synchronous FIFO and the signals shown in Figure 6.7 conform to its specification.

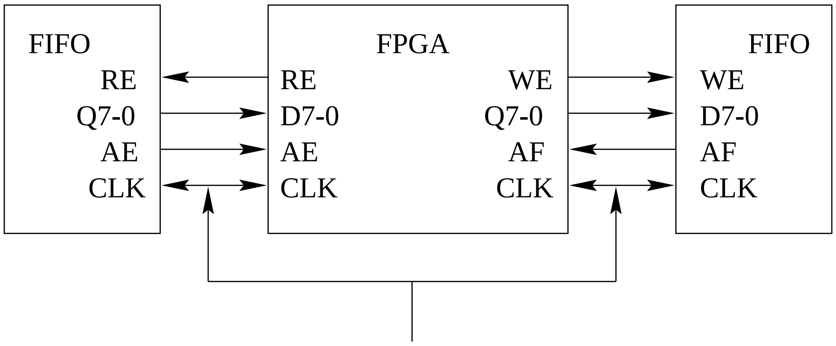

Figure 6.7: A FIFO buffer interface

The signals of the FIFO write interface are[4]:

1. Almost Full (AF) flag generated by the FIFO programmed to become active if the buffer does not have room for an entire cell. If the AF flag becomes active the processing element may finish writing the current cell after which it must stall and wait for the FIFO to become available again.

2. Write Enable (WE) signal generated by the FPGA.

3. Data (Q7-Q0) generated by the FPGA

4. Common clock used to synchronize signals to the FPGA and the FIFO.

The FIFO read interface the signals are:

[4]The names of the signals are sufficiently self descriptive so that the signal functions can be inferred. For more details, the reader is referred to the FIFO buffer datasheet[Cyp95].

1. Almost Empty (AF) flag generated by the FIFO programmed to become active when the processing element is reading the last cell. If the AF becomes active while reading the cell, the processing element may finish reading the current cell and then it must stop and wait until the AF flag becomes inactive again.

2. Read Enable (RE) signal generated by the FPGA.

3. Data (D7-D0) generated by the FIFO

4. Common clock for synchronization.

The FIFO interface is considered free of SALT attacks if it always generates the legitimate read and write cycles and maintains cell integrity. By the "legitimate cycle" we assume those specified by the device data sheet[Cyp95]. Theoretically one could design a test system which emulates the functionality of FIFO buffers and measures timing characteristics of the cycles generated by the FPGA device[5]. The test system would require a full behavioral model of the PE's interface to the FIFO buffer as well as the full model of the FIFO buffer. To test for timing requirements, a sampling rate significantly higher than the FPGA operating frequency would probably be needed, which would make it an extremely expensive solution.

To simplify the test procedure we have chosen not to scan for invalid read and write cycles. Instead, we scan for events on higher level that may be a result of an invalid cycle. We start by listing possible effects of invalid read and write cycles:

1. Read from an empty FIFO buffer.

2. Write to a full FIFO buffer.

3. Bogus read or write cycles.

4. Data corruption.

[5]Commercial products that automatically test the cycles on the I/O bus in a laboratory environment already exist[Hew]. Building an embedded version of a similar system specialized for the particular application would also be possible.

The first two cases can be easily caught by the data acquisition module. If, for example, the processing element starts writing a new cell (drives the WE active) when the AF is active, a write to a full buffer is detected. A similar case is reading from an empty buffer.

In the third case cell integrity will be violated which can be detected by preambles not being placed at the 49-byte boundary (see Section 3.2.2). To make detection more reliable a folded checksum[6] can be used in the upper four bits of the preamble (*i.e.*, a checksum over the previous cell could be calculated and used to generate a preamble for the next cell).

The fourth case will not necessarily violate the cell integrity, and may pass undetected by the methods we described so far. However, the higher layers of the protocol (*e.g.*, AAL-5 CRC-32 check) would easily detect this anomaly and drop the frame. In addition, if the cell integrity is always preserved then the effects of the attack are the same as those of software-like attacks (the third category in our classification) which is outside the scope of SALT attacks.

From the discussion above, we now summarize the tests that the system needs to perform on an isolated FPGA device before "approving" the device for use in a real system. The test vector generator connects to the FIFO buffer read interface and emulates the FIFO buffer. The processing element under test reads from the test vector generator and writes into the data acquisition module. Attempts to read from an empty buffer are caught by the test vector generator and an alarm is raised. The data acquisition module monitors the data for cell integrity and for attempts to write to a full buffer.

In addition we test if the configuration is a sink or a source. A device is considered a source if it generates an output cell stream without any input. Processing elements on the P4 may reduce or expand the occupied bandwidth but they may not generate a cell stream "out-of-nowhere". Likewise, a cell stream may not be completely sinked by the device.

These two properties are tested immediately after downloading the configuration into the test device. The test vector generator drives the AE signal active emulating the empty FIFO buffer, while the data acquisition module starts the timer and monitors the FIFO write interface. If the WE is pulled active before the timer expires, the configuration is

[6] Upper four bits XOR-ed with the lower four bits of an 8 bit checksum

declared a source and rejected. Next, the test vector generator drives the AE signal inactive for an extended period of time emulating the buffer with cells constantly available. The data acquisition module starts the timer and monitors the WE signal. If the timer expires before the WE signal is pulled active the device is declared a sink and rejected.

Finally, the test vector generator generates the stream of cells with various traffic patterns from constant traffic to bursty traffic and a random cell content, attempting to cover as many as possible states inside the processing element. Since the full functionality of the processing element is not known to the system, the test vector generator could only *attempt* to cover all states of the processing element by performing long tests.

The data acquisition module constantly monitors the write interface and searches for attempts to write to a full buffer, read from an empty buffer, and violations of cell integrity. Any such event will generate an alarm which will result in rejecting the configuration.

It can be observed from our case study that, in general, the tests for SALT attacks are *not* exhaustive and there is no formal proof that all illegal bus cycles will be detected. In fact, it is possible that an unsafe device will pass the test undetected, but the probability of this event can be reduced by performing longer and more complex tests.

The problem is analogous to the functional testing of a VLSI device where the success of a test depends how well the device functionality is covered by the test vectors. Here the problem is even harder because the device functionality is *not* known in advance and there is no specification to test against. Hence, common functional properties must be created *artificially* to allow the tests. For example, the artificial property we have created and enforced is the existence of a known bit pattern in the cell preamble. By testing for this property we were able to eliminate devices that may potentially violate cell integrity and confuse other elements of the system.

6.4 FPGA Security in P4

The P4 supports in part the concepts of FPGA security discussed in this chapter. The test for MELT attacks is a software module which parses through the configuration file and counts the number of outputs connected to each routing resource in the FPGA device.

Should any of these counts be greater than one, the configuration is declared unsafe and rejected. Since the information about the configuration file format of the Altera FLEX10K devices is not available[7], we were unable to identify the locations of bits that control the pass gates that connect the logic blocks to routing resources. Hence, we implemented the test as a routine that creates the same processing burden as the real test program would do, but parses randomly chosen bits in the device and ignores the result. The result is the function useful for testing the performance overhead introduced by the test routine, but without the ability to perform the actual test. The 90% confidence interval (in microseconds) for the time necessary to perform the MELT security test was measured to be $[3367.50, 3386.70]$. Measurements were taken on the same PC that controlled the P4 board in experiments described in Section 5.6.2. From the results shown in the Table 5.2, it follows that the adding the protection from MELT attacks, would increase T_{update} by roughly 30%. Depending on the number of FPGA configurations in the booster, the overall deployment latency could be increased by up to 15%.

The SALT attacks for the FIFO buffer interface are tested on a separate P4 board which is not connected to the network and which has the test vector generator and the data acquisition module in place of the switching array[8]. The controller initiates traffic generation, manipulates with timers and scans for alarms in the test system.

We have implemented the security tests only for the buffer interface part of the processing element. Since all buffer interface signals are unidirectional that part of the board level design is safe in the context discussed in Section 6.3.1. The controller interface part is not safe since it contains a bidirectional data bus. To make the P4 platform fully safe a new board will have to be built. Building a safe version of the P4 would be a good proof of concept. However, the feasibility of the design discussed in Section 6.3.1 is obvious and implementing a new board would not significantly enhance the research value of our project except for being a technical exercise. Therefore, we have left the implementation of safe P4 for future work.

[7]As mentioned earlier information, about the configuration file format is maintained as a trade secret by the FPGA vendors

[8]Since the switching array is also an FPGA device programmed at system initialization time, replacing it with the SALT testsystem is trivial.

We measured the time necessary to perform the test for the SALT attack described in Section 6.3.2. The 90% confidence interval (in microseconds) was measured to be $[T_{timeout}+67482, T_{timeout} + 99647]$, where $T_{timeout}$ is the value of the timer. The additive timer constant is due to the fact that in the first stage of the test, we must wait for the timer to expire before declaring that the device is not a source. In our experiments the timer was set to expire after 2 seconds. As opposed to the MELT test, the SALT test is an extremely costly operation and can increase the overall deployment latency by several orders of magnitude. Even if the source-test was not performed (*i.e.*, $T_{timeout} = 0$), the time to update the entry in the booster descriptor table is increased at least six times.

6.5 Comparison to Software Systems

The effect of the attack on systems that utilize reconfigurable logic is the ability to physically destroy the hardware. The only known attack on software systems with a similar effect is the CIH virus[CIH] that attacks a system by reprogramming the FLASH memory used to store the BIOS code. If the boot block of the FLASH is reprogrammed, the system may be unable to boot or could have an embedded malicious code that cannot be removed without the technician's intervention (*i.e.*, replacing or off-board reprogramming of the BIOS chip). Although the hardware is not actually destroyed in such an attack, the effect seen by the user is the same since the system becomes inoperable and unrecoverable.

The attack by the CIH virus can be made recoverable by using FLASH memory with the boot block protection feature[Int97]. If the boot block is write protected by the appropriate hardware, it is guaranteed to remain intact and presumably correct. Hence, if other parts of the BIOS become corrupted, the correctness of the boot block makes the recovery possible as discussed in [Arb98].

In FPGA attacks, the hardware destruction is actual, not apparent, and recovery without physically replacing the chip is not possible. Hence the class of attacks enabled by the use of reconfigurable logic is a completely new phenomena which will have to be well understood if computing systems based on reconfigurable logic are to be widely used.

Chapter 7

Conclusions and Future Work

The primary goal of this work was to demonstrate the application of reconfigurable hardware in building reconfigurable networks, the resulting performance benefits and the newly opened design space. In this chapter we highlight the points that illustrate that the research goal has been successfully achieved and point out other contributions of the research.

7.1 Summary of Presented Work

We observed two trends in computing and communications: convergence of these two fields into one and research efforts towards building reconfigurable systems in both fields. We also noted that network protocol processing contains a substantial number of specialized "bit-intensive" tasks whose performance can be improved by several orders of magnitude if these tasks are delegated to hardware. To accelerate such functions while maintaining flexibility, we built the Programmable Protocol Processing Pipeline (P4), a board based on run-time reconfigurable hardware which served as a building block for implementing a run-time reconfigurable network infrastructure. We experimentally verified the usefulness of the concept in the Protocol Boosters framework.

Using the Forward Error Correction example we discussed the benefits of adaptation, emphasized the need for hardware implementation and provided quantitative evaluation of the concept. We choose FEC as a simple and easy to understand case study which has the properties necessary to illustrate the research claims. Namely, the FEC falls into the class of bit-intensive protocol processing functions suitable for hardware implementation, performs best if used selectively, and can be transparently inserted in or removed from the protocol stack.

The adaptive system we analyzed, built and experimented with in Chapter 4 clearly shows that such adaptation is possible and beneficial in hardware. Moreover, the substantial processing overhead observed with software implementation of the same function implies that hardware implementation is highly desirable. We further argued that the FEC is not an isolated case, but exhibits the properties common to all bit-intensive protocol processing functions, such as data compression, encryption, *etc.*.

To cover the second class of reconfigurable networks, programmable (active) networks, we extended the model of adaptive protocol which allowed users to deploy their own protocol processing modules (*i.e.*, mechanisms) and algorithms for determining when the certain module is needed (*i.e.*, policy modules). This provided a programmable network infrastructure where the protocol processing algorithms are implemented in hardware allowing high performance in the data plane with the open programming model.

The fact that the user is allowed to program the hardware modules raises concerns about the security of such systems. If the system hardware may be programmed by the end user, the question arises if the attacker can be destructive in ways beyond that of software systems. We analyzed new opportunities for the attack and identified two new classes of threat unique to the systems that utilize reconfigurable logic. We further discussed the damages that the attacked system can incur and options that the designer has in making the system safe. We argued that the threat model of reconfigurable systems is a superset of that of software systems and the discussion focused on new classes of attack. Finally, we extracted the security concerns, a general issue present in all reconfigurable systems.

7.2 Contributions and Impact

Our work makes several contributions as listed in Section 1.3. All technical contributions (*i.e.*, hardware and software systems we have implemented) have been explained as we progressed in discussion of our work and in our previous publications [HS97], [HSM98], [Had98], [MHMS98], [HMS99], [HUS99]. In this section we provide the final elaboration on the three major scientific contributions of our work: applying reconfigurable computing to reconfigurable networking, reconfigurable systems security and completion and extension of the Protocol Boosters model.

7.2.1 Design Space for Reconfigurable Networks

Many researchers have questioned the performance implications of programmable and adaptive network infrastructure arguing that flexibility must be provided at the expense of performance. The available bandwidth of state-of-the-art networks is several orders of magnitude higher than the bandwidth that can be utilized by a state-of-the-art general purpose CPU. Increasing the processing burden on the network node, by making the system flexible, could increase the performance gap even further and drastically degrade overall system performance. One way to cope with the performance loss is to restrict the programmability to the network "control plane". Implementing a programmable control infrastructure on top of a classical network is an interesting tradeoff between flexibility and performance. The resulting network is more flexible than the classically designed network with little impact on performance.

The approach we have taken allows us to address the performance problems without the need to restrict the programmability to the control plane. This is a new approach which complements previous research directions and creates a new dimension in the design space. Our system provides the following tradeoff: Programmable hardware can accommodate a limited number of processing functions which execute several orders of magnitude faster than their software counterparts. However the number and the complexity of functions that can take the fast path is limited by the available programmable hardware resources. In addition to balancing between the amount of flexibility and system performance, the user can attempt to select an optimal subset of functions that can be delegated to hardware. If more functions can take the fast path either by being used more often or by providing more available resources, a higher degree of programmability in the data plane can be achieved without a substantial loss in performance.

7.2.2 Security in Reconfigurable Computing

Reconfigurable computing fills the gap in the design space where the nature of algorithms is such that the hardware implementation is essential, without loss of generality of the architecture. Many researchers have studied mapping various algorithms to reconfigurable logic, design of reconfigurable computing platforms and the programming models for reconfigurable computing. Their results paved the way towards making reconfigurable logic an integral part of a general purpose computing systems.

The fact that reconfigurable logic can be attacked at a very low level, allowing physical destruction of the hardware, will become a major obstacle in using reconfigurable systems in a secure manner. Systems built from the combination of a general purpose CPU and reconfigurable logic could allow the construction of an extremely destructive computer virus that uses software to implement the replication mechanism and performs its destructive action in hardware. Such a virus would be far more destructive than any software virus while its replication mechanism could have the same complexity.

We have demonstrated that this threat is realistic, and that the physical destruction of the hardware is possible. In addition, we made a constructive contribution outlining the methods of attack prevention and detection. We have shown that building a safe runtime environment is possible, provided that the threat is well understood and seriously addressed. As run-time reconfigurable systems become widely adopted, the security issues will become more important and we believe that our work is a start of a new field which combines research in computer security with that of reconfigurable computing.

7.2.3 Completion and Extension of Protocol Boosters Model

The Protocol Boosters model represents an interesting way to analyze and build adaptive protocols. Prior to our work, Protocol Boosters research has concentrated on demonstrating a variety of mechanisms, and implementing software prototypes. The major weakness was the inability to provide a convincing example of a sufficiently complex policy that demonstrates the usefulness of the concept. The policy was considered a secondary issue, and attention was focused on providing a kernel level implementation of a modular protocol stack[MMR98] that would support the concept.

This thesis argues that the most important and the most challenging aspect of adaptive protocol design is the policy, not the mechanism[HMS99]. We provided an example of the first operational policy module, which thus completed the Protocol Boosters picture. The experiments have demonstrated a substantial performance improvement compared to the original TCP/IP protocol, and extended the operating range of the protocol in the face of bit errors. In addition we have demonstrated that the same concept is realizable in hardware, which is a significant extension of the original model. Extending the design space to hardware reduced the mechanism processing overhead, allowing operation at high performance.

Finally, we extended the model to provide end user programmability. Protocol Boosters can be viewed as an application of programmable networks or as a road towards a fully programmable network infrastructure. Our work has shown how the adaptive model can be easily extended to a programmable model.

7.3 Future Directions

There are two main directions of possible future work: further development of our concepts from the networking point of view and from the reconfigurable computing point of view.

From the networking point, concepts similar to P4 can be further developed and specialized for various types of networking equipment. An architecture similar to the P4 can be integrated better with hosts, line cards and port controllers on switches and routers. This would provide a tight coupling between the programmable hardware and the rest of the system unlike the current P4, and other work done at Washington University[Dec99].

For example, on the host side, the P4-like board could be a combination of a line card (like the present version of the P4) and a host accelerator. The extension could be easily done by allowing the reconfigurable board to accept and deliver traffic on the host bus. The card could have access to buffers in the host memory and be able to process the traffic between the network and the host as well as network-to-network data and local host data. This would extend the set of functions that can be efficiently implemented in programmable accelerators and provide the migration to a general processing model discussed in Section

5.1 (Figure 5.1 - right). Alternatively, a switch with a CPU-based processing engine in a switching fabric[Mar96] could be augmented with programmable logic resulting in the tightly coupled combination of a CPU, programmable logic and a switch.

It would also be desirable to move the processing to a higher level and eliminate the ATM dependency of the P4. To achieve this, the packet processing should be viewed as processing a block of user data (*i.e.*, payload) without exposing the particular network infrastructure. The concepts learned from the P4 are general and ATM was used as a convenience, not as a necessary technology.

Compared to the memory of a CPU based computing system, the programmable logic devices represent a very limited set of processing resources. It is possible that the number of available processing elements is smaller than the number of functions simultaneously required. In such cases, some functions would have to take the slow path and execute in software. Choosing the best set of functions to be mapped to programmable logic for improved performance is a multidimensional optimization problem. The choice of a device to be overwritten when the new processing function is downloaded (*i.e.*, replacement policy) can also affect the system performance. Also, if there are multiple boosters present on the node, there will be greater contention for hardware resources among the policy threads. In its current state, the system provides no arbitration logic and the resources are assigned to a booster on a first come, first served basis. With an increasing number of boosters, more advanced arbitration logic would be necessary to resolve the contention problem. Finally, successfully *predicting* the need for a processing algorithm can improve the system agility. The problems we listed fall into *resource management*, which has not been the focus of our work. For efficient utilization of programmable logic, resource management will need to be well understood and studied.

From the reconfigurable computing perspective, our work has covered two aspects. First, it provided another example application of the technology which can be used to demonstrate the usefulness of reconfigurable computing. Second, the security topic can be studied without focusing on networking applications. The problem we have identified is general, and opens a new area with a variety of opportunities for future work.

The configuration file analysis is a powerful method that can prevent the most destructive form of attack. Possible future work could extend these methods to other programmable logic families. The Altera FLEX8000[Alt98c] and FLEX10K[Alt98d] families use a simple routing network and the complexity of the analysis algorithm is low. Other programmable logic families have much more complex routing networks which may consist of short routing lines with crossbar switches at the intersections. In such cases, the resulting routing path may have an arbitrary shape which dramatically complicates the analysis algorithm.

Finally, a challenging research direction is providing providing a higher level programming model for platforms similar to the P4. This would not only simplify the construction of a user's protocol processing functions, but also enhance system security. Although the functions are programmed in high-level programming languages such as VHDL, the low-level concerns such as signal timing and synchronization, delays and jitter etc., still must be addressed by the programmer which inhibits, software engineers from using the programmable logic. The programming model can be improved with a hierarchical device architecture where the programmer sees a coarse grain structure and has access to functional blocks that hide the low level details, making the design resemble classical programming in a highly parallel environment. Providing a high level programming environment would improve system security as formal verification methods may become applicable, while the placement and routing is simplified with a coarse grain architecture.

Appendix A

Discussion on Packet and Bit Error Rate

The analysis of Section 4.3 assumes that the packet error indicator (denoted as $r[k]$) is an array of independently distributed Bernoulli random variables. The Network Impairment Emulator[Dim97] generates bit errors with an exponentially distributed time between two errors, where the mean number of bytes between errors is a parameter used to program the emulator. In this section, we establish the relationship between packet errors used to analyze the policy module and bit errors generated by the Network Impairment Emulator.

Bit errors can be viewed as events whose interarrival time t_i has the exponential distribution $\epsilon(\lambda)$ as shown in Figure 8.1.

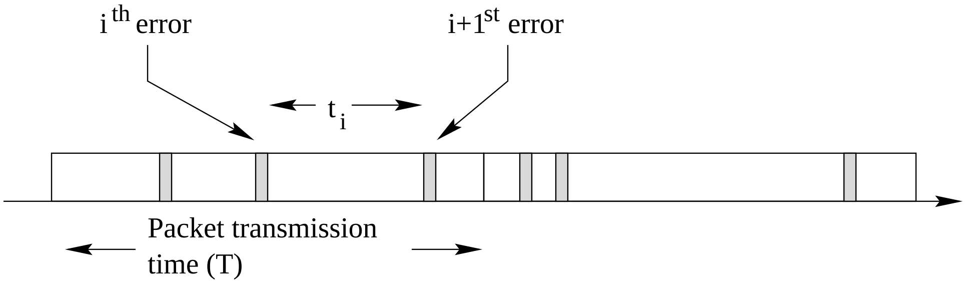

Figure 8.1: Exponentially distributed bit errors

The parameter λ is the bit error frequency in errors per second and the probability density function of the error interarrival time is:

$$f_t(t) = \lambda \cdot e^{-\lambda \cdot t} \tag{8.1}$$

As a prerequisite for our discussion, we state two theorems without proof. The proof can be found in [Chu74].

Theorem 1 *Consider the sequence of events whose interarrival times are positive random variables with the exponential distribution $\epsilon(\lambda)$. The total number of arrivals in a time interval of length T has the Poisson distribution $\pi(\lambda \cdot T)$.*

Theorem 2 *If the arrivals are counted over disjoint intervals, the resulting random variables are independent.*

Assuming that the packet transmission time is T, it follows from Theorem 1 that the number of bit errors in the packet has the Poisson distribution. Hence, the probability of having n errors in the packet is:

$$P_e(n) = \frac{(\lambda \cdot T)^n}{n!} e^{-\lambda \cdot T} \tag{8.2}$$

The parameter $\lambda \cdot T$ is the packet error rate in errors per packet. The packet is in error if it contains *at least* one bit error. Therefore, the probability of packet error can be written as:

$$P_{packet} = 1 - P_e(0) = 1 - e^{-\lambda \cdot T} = \lambda \cdot T - \frac{(\lambda \cdot T)^2}{2!} + \frac{(\lambda \cdot T)^3}{3!} - \ldots \approx \lambda \cdot T \tag{8.3}$$

The last approximation in Equation (8.3) is valid for a small packet error rate, since the higher order terms are negligible.

By definition, the packet error indicator $r[k]$ equals 1 when the packet is in error and 0 otherwise. Clearly this is a Bernoulli random variable with parameter $\lambda \cdot T$. Since packets do not overlap in time, it follows from Theorem 2 that packet errors are independent.

Finally, the time to transmit one packet can be written as a product of the time to transmit one bit and the number of bits in the packet. Hence, the packet error rate can be written as:

$$P_{packet} = L \cdot T_{bit} \cdot \lambda = L \cdot P_{bit} \tag{8.4}$$

where P_{bit} is the probability of bit error (or bit error rate) and L is the size of the packet in bits.

In summary, the Network Impairment Emulator generates exponentially distributed bit errors which results in independent packet errors with a rate of $L \cdot P_{bit}$.

Appendix B

Cell Level vs. Packet Level Decoding

In Section 4.2.2, we mentioned that software implementation of the Viterbi decoder has a coding advantage over the hardware implementation because it operates on the packet level instead of a cell level. In this section we review the principles of Viterbi decoding and explain the advantage of decoding on the packet level. We use the decoder example discussed in [PS94] to illustrate how resetting causes the decoder to fail in correcting, otherwise correctable, bit errors.

The basic principle of Viterbi decoding is tracing a trellis diagram. Each node in the graph can branch up or down and the branching direction corresponds to input data bits, 0 or 1 respectively. Each branch has a metric assigned to it which represents the encoder output bits. The complexity of a graph and branch metrics are defined by the actual code: coding rate, constrain length and encoding polynomials.

Figure 9.1 shows an example trellis diagram and the path followed when encoding a data sequence 101100 (the corresponding path is down-up-down-down-up-up). The resulting encoded sequence is 110100101011 (see branch metrics). Decoding consists of expanding all possible paths for the received bit sequence and calculating the accumulated Hamming distance between the received bit-sequence and branch metrics. The path with the minimum accumulated Hamming distance is assumed to correspond to decoded bit sequence.

To illustrate the effects of resetting the decoder, assume that the first two bits of the encoded sequence are in error. Hence, the decoder receives 000100101011. Figure 9.2 shows the surviving paths and their accumulated Hamming distances after the complete

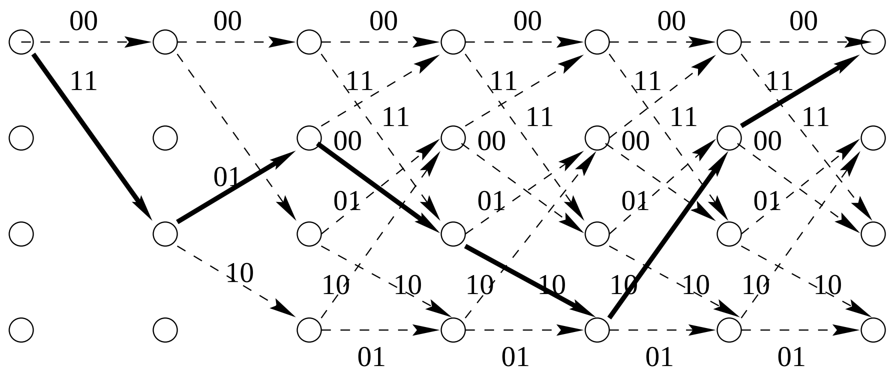

Figure 9.1: Trellis diagram for the data sequence 101100

sequence has been received. The minimum accumulated Hamming distance is 2 and the corresponding path is correctly decoded into the original bit sequence (101100): the double-bit error has been corrected.

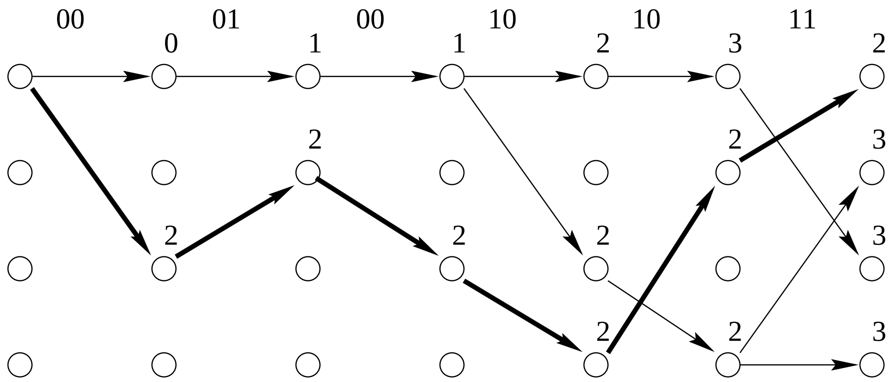

Figure 9.2: Surviving paths and respective accumulated Hamming distances for the encoded sequence 000100101011 (decoder does not reset)

Now assume that after the first half of the sequence the decoder and the encoder are restarted. The corresponding path in the trellis graph is shown in Figure 9.3.

The encoded sequence is now: 110100110111. Assuming the same double-bit error as in the previous example, the decoder receives 000100110111. However, the decoder now resets after receiving half of the sequence and assumes that the correct path is the one corresponding to the bit sequence 000, as shown in the Figure 9.4. The second half is decoded properly and the resulting decoded sequence is 000100.The incorrect result is a direct consequence of resetting the decoder.

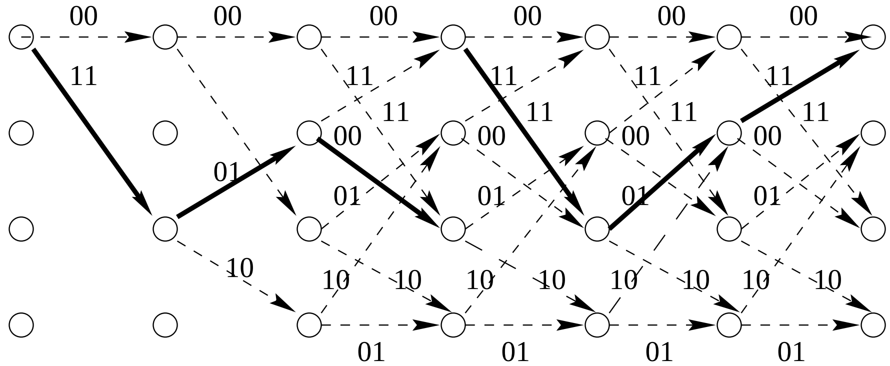

Figure 9.3: Trellis diagram for the data sequence 101100. Encoder resets in the middle of the sequence

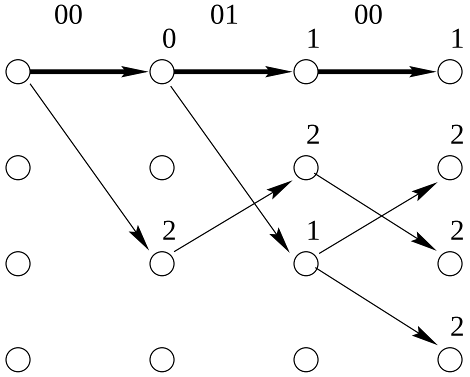

Figure 9.4: Surviving paths and accumulated Hamming distances before resetting the decoder. Sequence incorrectly decoded to 000

In practice, the decoder is implemented as the buffer which stores the paths and the add-compare-select units which select surviving paths. If the buffer is deeper, decoding is more reliable because the decoder has an increased chance to eliminate non-surviving paths before it must make a decision. When the decoder is reset, the buffer must be flushed and the path with a minimum accumulated Hamming distance is selected. The decoding is unreliable for the bits that most recently entered the buffer, and the more frequently we reset the decoder, the higher the probability of incorrect decoding.

Since the software implemented decoder operates on the packet level and thus resets less frequently than its hardware counterpart (which operates on the cell level), the coding advantage can be observed for high bit error rates.

Bibliography

[ABD92] J. M. Arnold, D. A. Buell, and E. G. Davis. Splash 2. In *Proceedings of the 4th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–324, June 1992.

[Ale98] D. S. Alexander. *ALIEN: A Generalized Computing Model of Active Networks.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, October 1998.

[Ale98a] D. S. Alexander, *et al.* The SwitchWare Active Network Architecture. *IEEE Network Special Issue on Active and Controllable Networks*, 12(3):29–36, 1998.

[All97] P. Allke. *Configuration Issues: Power-up, Volatility, Security, Battery Back-up.* Xilinx Inc., November 1997. Application Note 092, Version 1.1.

[Alt98a] Altera Corporation. *Altera Device Package Information - Data Sheet*, March 1998. Version 7.

[Alt98b] Altera Corporation. *Configuring FLEX10K Devices, Application Note 59*, August 1998. Version 1.01.

[Alt98c] Altera Corporation. *Flex 8000 Programmable Logic Family - Data Sheet*, 9.11 edition, September 1998.

[Alt98d] Altera Corporation. *FLEX10K Embedded Programmable Logic Family - Data Sheet*, October 1998. Version 3.13.

[Arb98] W. A. Arbaugh. *Chaining Layered Integrity Checks.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, December 1998.

[ASNS97] D. S. Alexander, M. Shaw, S. M. Nettles, and J. M. Smith. Active Bridging. In *Proceedings of ACM Sigcomm 97*, September 1997.

[ATM] ATM on Linux home page and source code. URL `http://lrcwww.epfl.ch/linux-atm/`.

[ATM94] ATM Forum. *ATM User-Network Interface, Version 3.1 (UNI 3.1) Specification*, May 1994. see also `ftp://ftp.atmforum.com/pub/approved-specs/af-uni-0010.002/`.

[BC94] Computer Science & Telecommunications Board and National Research Council, editors. *Academic Careers for Experimental Computer Scientists and Engineers.* National Academy Press, 1994. ISBN 0-309-04391-8.

[BDH+97] J. Burns, A. Donlin, J. Hogg, S. Sinhh, and M. de Witt. A Dynamic Reconfiguration Run-Time System. In *Proceedings of 5th IEEE Symposium on FPGAs for Custom Computing Machines (FCCM'97*, April 1997.

[BPSK96] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz. A Comparison of Mechanisms for Improving TCP Performance over Wireless Links. In *Proceedings of ACM Sigcomm 96*, pages 256–269, August 1996.

[Chu74] K. L. Chung. *Elementary Probability Theory with Stohastic Processes*, chapter 7.2: Poisson Process, pages 199–210. Springer-Verlag, 1974.

[CIH] Global CIH Virus Information Center. Information on the Web. URL `http://www.datafellows.com/cih/`.

[Cor99] 3COM Corporation. Introducing the 3CR990-TX-97 10/100 PCI NIC with 3XP, 1999. URL `http://www.3com.com/products/dsheets/3cr990.html`.

[CWG+98] S. Cadambi, J. Weener, S. C. Goldstein, H. Schmit, and D. E. Thomas. Managing Pipeline-Reconfigurable FPGAs. In *Proceedings of ACM Sixth International Symposium on FPGAs*, February 1998.

[Cyp95] Cypress Semiconductor Company. *CY7C4251 Synchronous FIFO - Data Sheet*, March 1995. Revised, November 1997.

[Dal93] C. Dalton, *et al.* Afterburner. *IEEE Network Magazine*, 7(4):36–43, July 1993.

[Dec99] D. S. Decasper, *et al.* A Scalable High-Performance Active Network Node. *IEEE Network*, 13(1):8–19, January/February 1999.

[Dim97] R.W. Dmitroca, *et al.* Emulating ATM Network Impairments in the Laboratory. *Hewlett-Packard Journal*, 48(2):45–50, April 1997.

[DH98] S. Deering and R. Hinden. Internet Protocol. Internet RFC 2460, December 1998.

[DNSS94] N. Dagdeviren, J. A. Newell, L. A. Spindel, and M. J. Stefanick. Global Networking with ISDN. *IEEE Communications Magazine*, 32(6):26–32, June 1994.

[Ele] Electronic Design Interface Format (EDIF) Information on the Web. URL `http://www.edif.org`.

[FMS+98] D. C. Feldmeier, A. J. McAuley, J. M. Smith, , D. Bakin, W. S.Marcus, and T. Raleigh. Protocol Boosters. *IEEE JSAC Special Issue on Protocol Architectures for the 21st Century*, 16(3):437–444, April 1998.

[For] Linux device driver for Fore PCA200 adaptor, home page and source code. URL
`http://os.inf.tu-dresden.de/project/atm/`.

[GN96] R. Gilligan and E. Nordmark. Transition Mechanism for IPv6 Hosts and Routers. Internet RFC 1933, April 1996.

[Gok91] M. Gokhale, *et al.* Building and Using a Highly Parallel Programmable Logic Array. *IEEE Computer*, 24(1):81–89, January 1991.

[Had98] I. Hadžić. Signalling Protocol for P4 (SPP4), version 1.0. Technical Report MS-CIS-98-20, University of Pennsylvania, CIS Department, 1998.

[Hew] Hewlett-Packard Corporation. *HP E2925B 32bit, 33MHz and HP E2926B 32/64bit, 33MHz PCI Exerciser & Analyzer*. Technical Specifications, URL `http://www.tm.agilent.com/tmo/datasheets/English/HPE2925B.html`.

[HHW90] R. W. Hartenstein, A. G. Hirschbiel, and M. Weber. XPUTERS: An Open Family of non-Von Neumann Architectures. In *Proceedings of 11th ITG/GI - Conference Architectur von Rechensystemen*. VDE/Verlag, 1990. see also `http://xputers.informatik.uni-kl.de`.

[Hic98] M. W. Hicks. *PLAN Service Programmer's Guide*. University of Pennsylvania, 1998. URL `http://www.cis.upenn.edu/~switchware/PLAN/docs-ocaml/serv_guide/index.html`.

[Hic99] M. W. Hicks, *et al.* PLANet: An Active Internetwork. In *Proceedings of IEEE Infocom 99*, March 1999.

[HKM+98] M. W. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[HL87] J. Hagenauer and E. Lutz. Forward Error Correction Coding for Fading Compensation in Mobile Satellite Channels. *IEEE JSAC*, 5(2):215–225, February 1987.

[HMS99] I. Hadžić, W. S. Marcus, and J. M. Smith. Policy and Mechanism in Adaptive Protocols. Technical Report MS-CIS-9903, University of Pennsylvania, Department of Computer and Information Science, 1999.

[HP91] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[HS97] I. Hadžić and J. M. Smith. P4: A Platform for FPGA Implementation of Protocol Boosters. In *Proceedings of 7th International Workshop on Field-Programmable Logic and Applications, FPL'97*, LNCS 1304, pages 438–447. Springer, September 1997.

[HSM98] I. Hadžić, J. M. Smith, and W. S. Marcus. On-the-fly Programmable Hardware for Networks. In *Proceedings of IEEE Globecom 98*, November 1998.

[HUS99] I. Hadžić, S. Udani, and J. M. Smith. FPGA Viruses. In *Proceedings of 9th International Workshop on Field-Programmable Logic and Applications, FPL'99*, LNCS. Springer, August 1999.

[HW98] E. K. Hall and S. G. Wilson. Design and Analysis of Turbo Codes on Rayleigh Fading Channels. *IEEE JSAC*, 16(2):160–174, 1998.

[IEE93] IEEE Standard. *VHDL Language Reference Manual*, 1993. IEEE Std. 1076-1993 - revision of IEEE Std. 1076-1987.

[IEE95] IEEE Standard. *Hardware Description Language based on Verilog HDL*, 1995. IEEE Std. 1364-1995.

[Int93] International Telecommunications Union (ITU). *AAL Type 5, Draft Recommendation Text for Section 6 of I.363*, January 1993.

[Int97] Intel Corporation. *1-MBit (128K × 8) Boot Block FLASH Memory - Data Sheet*, December 1997.

[Jac88] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of ACM Sigcomm 88*, August 1988.

[JGB+90] N. D. Jones, C. K. Gomrad, A. Bondirf, O. Danvy, and T. Mongensen. A Self-Applicable Partial Evaluator for the Lambda Calculus. In *Proceedings of IEEE International Conference on Computer Languages*, March 1990.

[JPS85] N. D. Jones, P.Sestift, and H. Sondergaard. An Experiment in Partial Evaluation: The Generation of a Compiler Generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications*, volume 202 of *LNCS*. Springer-Verlag, 1985.

[JW97] J. Jamison and R. Wilder. vBNS: The Internet Fast Lane for Research and Education. *IEEE Communications Magazine*, 35(1):60–63, January 1997.

[KHN97] R. Kress, R. W. Hartenstein, and U. Nageldinger. An Operating System for Custom Computing Machines based on Xputer Paradigm. In *Proceedings of 7th International Workshop on Field-Programmable Logic and Applications*, volume 1304 of *LNCS*, pages 304–313, 1997.

[KSZ95] K. Kleinpaste, P. Steenkiste, and B. Zill. Software Support for Outboard Buffering and Checksumming. In *Proceedings of ACM Sigcomm 95*, pages 87–98, October 1995.

[LBL95] A. A. Lazar, S. K. Bhonsle, and K. S. Lim. A Binding Architecture for Multimedia Networks. *Journal on Parallel and Distributed Systems*, 30(2):204–216, November 1995.

[LC83] S. Lin and D. J. Costello. *Error Control Coding*, chapter 3, 10 and 11. Prentice-Hall, 1st edition, 1983.

[Ler99] Xavier Leroy. *The Objective Caml System, Documentation and User's Guide*. INRIA, 1999. URL
`http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`.

[LJMA99] D. C. Lee, M. T. Jones, S. F. Midkiff, and P. M. Athanas. Towards Active Hardware. In *Proceedings of International Workshop on Active Networks*, LNCS 1653, pages 180–187, June 1999.

[LLM96] A. A. Lazar, K. S. Lim, and F. Marconcini. Realizing a Foundation for Programmability of ATM Networks with the Binding Architecture. *IEEE JSAC*, 14(7):1214–1227, September 1996.

[LR99] Xavier Leroy and François Rouaix. *Security properties of typed applets*, pages 147–182. LNCS 1603. Springer, 1999.

[LWG98] U. Legedza, D. Wetherall, and J. Guttag. Improving the Performance of Distributed Applications Using Active Networks. In *Proceedings of IEEE Infocom 98*, April 1998.

[LY82] S. Lin and P. S. Yu. A Hybrid ARQ Scheme with Parity Retransmission for Error Control on Satellite Channels. *IEEE Transactions on Communications*, 30(7):1701–1719, July 1982.

[LY83] S. Lin and P. S. Yu. A Modified Selective-Repeat Type II Hybrid ARQ System and its Performance Analysis. *IEEE Transactions on Telecommunications*, 31(5):593–607, May 1983.

[Man97] W. H. Mangione-Smith, *et al.* Seeking Solutions in Configurable Computing. *IEEE Computer Magazine*, 30(12):38–43, December 1997.

[Mar96] W. S. Marcus. An Architecture for QoS anaysis and experimentation. *IEEE Transactions on Networking*, 4(4):597–603, August 1996.

[MCS97] A. Mallet, J. D. Chung, and J. M. Smith. Operating Systems Support for Protocol Boosters. In *Proceedings, HIPPARCH Workshop*, June 1997.

[Mel93] T. F. Melham. *Higher Order Logic and Hardware Verification.* Cambridge University Press, 1993.

[MHMS98] W. S. Marcus, I. Hadžić, A. J. McAuley, and J. M. Smith. Protocol Boosters: Applying Programmability to Network Infrastructures. *IEEE Communications Magazine*, 36(10):79–83, October 1998.

[Min89] S. E. Minzer. Broadband ISDN and Asynchronous Transfer Mode (ATM). *IEEE Communications Magazine*, 27(9):17–24,57, September 1989.

[MMR98] W. S. Marcus, A. J. McAuley, and T. Raleigh. Protocol Boosters: A Kernel-Level Implementation. In *Proceedings of IEEE Globecom*, November 1998.

[MSK+98] T. Miyazaki, K. Shirakawa, M. Katayama, T. Murooka, and A. Takahara. A Transmutable Telecom System. In *Proceedings of 8th International Workshop on Field Programmable Logic and Applications*, LNCS 1482, pages 366–375. Springer, September 1998.

[Nec97] G. C. Necula. Proof-Carrying Code. In *Proceedings of The 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, January 1997.

[OP92] S. W. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[PFTK98] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. In *Proceedings of ACM Sigcomm 98*, August 1998.

[PMC95] PMC-Sierra, Inc. *S/UNI-155-LITE Optical Reference Design (SORD)*, September 1995. Issue 2.

[PMC96] PMC-Sierra, Inc. *S/UNI-155-LITE, Data Sheet*, March 1996. Issue 6.

[Pos81a] J. Postel. Internet Protocol. Internet RFC 791, September 1981.

[Pos81b] J. Postel. Transmission Control Protocol. Internet RFC 793, September 1981.

[PS94] J. G. Proakis and M. Salehi. *Communication Systems Engineering*, chapter 10.7: Convolutional Codes, pages 777–792. Prentice Hall, 1994.

[RF94] A. Romanow and S. Floyd. Dynamics of TCP Traffic over ATM Networks. In *Proceedings of ACM Sigcomm 94*, volume 24, pages 79–88, October 1994.

[Riv92] R. Rivest. MD-5 Message-Digest Algorithm. Internet RFC 1321, April 1992.

[SA95] T. Shanley and D. Anderson. *ISA System Architecture*, chapter 16 and 17: ISA Bus Structure and Types of ISA Bus Cycles, pages 335–364. Addison-Wesley, 1995.

[Say96] Z. Sayeed. *Adaptive Coding and Transmitter Diversity for Slow Fading Channels*. PhD thesis, University of Pennsylvania, Department of Electrical Engineering, December 1996.

[Sch99] B. Schwartz, *et al.* Smart Packets for Active Networks. Presented at OpenArch'99, March 1999. see also `http://www.net-tech.bbn.com/smtpkts/smtpkts-index.html`.

[SJ89] C. Schlegel and D. J. Costello Jr. Bandwidth Efficient Coding for Fading Channels: Code Construction and Performance Analysis. *IEEE JSAC*, 7(9):1356–1368, December 1989.

[SM92] S. Soliman and K. Mokrani. Performance of Coded Systems over Fading Dispersive Channels. *IEEE Transactions on Communications*, 40(1):51–59, January 1992.

[Smi99] J. M. Smith, *et al.* Activating Networks: A Progress Report. *IEEE Computer*, 32(4):32–41, April 1999.

[Tan96] A. S. Tanenbaum. *Computer Networks*, chapter 2.5.4 Perspective of N-ISDN, pages 143–144. Prentice Hall, 3rd edition, 1996.

[TSE98] D. Thomas, H. Schmit, and T. Eagan. Cached Virtual Hardware for Configurable Computing. FY-97 DARPA Progress Report, 1998. URL `http://www.darpa.mil/ito/psum1998/E263-0.html`.

[TSS+97] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[Tur86] J. S. Turner. New Directions in Communications (or Which Way to the Information Age). *IEEE Communications Magazine*, 24(10):8–15, October 1986.

[WE93] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.

[WGT98] D. J. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *Proceedings of IEEE Openarch'98*, April 1998.

[Xil97] Xilinx Inc. *XC6200, Data Sheet*, April 1997. Version 1.10.

[Xil99] Xilinx Inc. *XC4000E and XC4000X Series Field Programmable Gate Arrays, Data Sheet*, May 1999. Version 1.6.

Index