

SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors

A Thesis
Presented to
The Academic Faculty

by

Ada Gavrilovska

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

College of Computing
Georgia Institute of Technology
July 2004

Copyright © 2004 by Ada Gavrilovska

SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors

Approved by:

Karsten Schwan, Advisor

Peter Steenkiste
(Carnegie Mellon University)

Kenneth Mackenzie

Calton Pu

George Riley

Date Approved: 12 July 2004

ACKNOWLEDGEMENTS

This work would not have been possible without the support and sacrifice of many others. I would like to express my gratitude to them all.

First of all, I am in eternal debt to my advisor Karsten Schwan. His advise and guidance, the support and opportunities he gave me throughout my Ph.D. years are of invaluable importance. I would also like to acknowledge the other members of my thesis committee. I am grateful to Kenneth Mackenzie for the boost he gave to the IXP-based research at Tech, the code base he developed, which is utilized in the implementation of this work, and the many questions he always so readily raised. Calton Pu and George Riley challenged me and provided me with invaluable feedback from two very different perspectives. I would finally like to thank Peter Steenkiste for gracefully accepting to be part of this committee and for providing me with many insightful comments on this work and its presentation.

The College of Computing at Georgia Tech, and particularly the Systems Group, gave me the pleasure of meeting and working with many very talented students and researchers over the years. I would like to acknowledge Greg Eisenhauer, Austen McDonald, Ivan Ganey, Sanjay Kumar, Haile Seifu and Josh Fryman among others, who helped with many technical issues in this work. Special thanks to many past and present members of the Pizza&Birra crowd. Our friendship and discussions made these years an enjoyable journey.

But most of all, I would like to give my deepest thanks to my family, my husband, my brother, and especially my parents. Their love, support, sacrifice and encouragement gave me the strength and inspiration to make it through. This work is dedicated to all of them.

TABLE OF CONTENTS

| | |
|---|-----------|
| ACKNOWLEDGEMENTS | iii |
| LIST OF TABLES | viii |
| LIST OF FIGURES | ix |
| SUMMARY | x |
| CHAPTER 1 INTRODUCTION | 1 |
| 1.1 Background | 1 |
| 1.2 Motivation | 3 |
| 1.3 Thesis Statement | 4 |
| 1.4 SPLITS and Stream Handlers | 4 |
| 1.5 Organization | 7 |
| CHAPTER 2 HOST - ATTACHED NETWORK PROCESSOR PAIRS | 9 |
| 2.1 Network Processors | 9 |
| 2.1.1 Intel IXP Network Processors | 10 |
| 2.2 Host-ANP pairs: High Level View | 11 |
| 2.2.1 Data Path Through the Host-ANP Node | 12 |
| CHAPTER 3 APPLICATION DOMAIN | 14 |
| 3.1 Streaming Applications | 14 |
| 3.2 Network Processors and Application-specific Services | 16 |
| 3.3 Sample Applications Using Host-ANP Pairs | 18 |
| 3.3.1 Operational Information Systems: Delta Airlines | 18 |
| 3.3.2 Scientific Collaborations: The SmartPointer Framework | 21 |
| CHAPTER 4 PROGRAMMING MODEL | 24 |
| 4.1 Model Overview | 24 |
| 4.2 Basic Abstractions | 25 |
| 4.2.1 Data Streams and Services | 29 |
| 4.3 Activation Points | 29 |
| 4.4 Concrete Implementation of the Model | 33 |
| 4.5 Relationships with Other Models | 34 |

| | | |
|------------------|---|-----------|
| 4.5.1 | Models for Network-level Services | 34 |
| 4.5.2 | Models Used with Streaming Applications | 36 |
| CHAPTER 5 | STREAM HANDLERS | 38 |
| 5.1 | Concept and Definitions | 38 |
| 5.1.1 | Accessing Application-level Data | 40 |
| 5.2 | Stream Handler Implementation | 41 |
| 5.3 | Stream Handler Invocation | 44 |
| 5.3.1 | Classification | 44 |
| 5.3.2 | Enabling Activation Points | 45 |
| 5.3.3 | Receive Contexts | 47 |
| 5.3.4 | Transmit Contexts | 49 |
| 5.3.5 | Memory-resident Contexts | 51 |
| 5.3.6 | Resource Requirements | 53 |
| 5.4 | Programming Interface | 54 |
| 5.5 | Summary | 56 |
| CHAPTER 6 | SPLITS - SOFTWARE ARCHITECTURE FOR PROGRAM- | |
| | MABLE LIGHTWEIGHT STREAM HANDLERS | 58 |
| 6.1 | SPLITS Framework | 58 |
| 6.2 | System Components | 59 |
| 6.3 | SPLITS on Host-IXP1200 Nodes | 61 |
| 6.3.1 | Data Buffers | 62 |
| 6.3.2 | Data Management | 62 |
| 6.3.3 | Control Buffers | 63 |
| 6.3.4 | Control Management | 64 |
| 6.3.5 | Host-side Components | 66 |
| 6.3.6 | Deploying Stream Handlers onto the Host-ANP Data Path | 66 |
| 6.4 | SPLITS API | 66 |
| 6.5 | Dynamic Reconfiguration | 68 |
| 6.5.1 | Handler Selection | 70 |
| 6.5.2 | Parameter Reconfiguration | 71 |
| 6.5.3 | Dynamic Reloading | 71 |

| | | |
|---|--|------------|
| 6.6 | Summary | 72 |
| CHAPTER 7 SPLITS SUPPORT TOOLS | | 74 |
| 7.1 | Constraint Verifier | 74 |
| 7.2 | Control Manager | 77 |
| 7.3 | Resource Monitoring | 77 |
| 7.4 | Stream Handler Profiling | 79 |
| 7.5 | Safety, Security, Code Generation | 80 |
| CHAPTER 8 EVALUATION | | 82 |
| 8.1 | Experimental Setup | 82 |
| 8.2 | Improved Overlay Network Performance Using Host-ANP Nodes | 83 |
| 8.2.1 | ANP-forwarding Reduces Latency and Improves Throughput | 83 |
| 8.2.2 | Importance of Multiple Activation Points on ANPs | 85 |
| 8.3 | Application-specific Services on ANPs: Feasibility and Limitations | 87 |
| 8.3.1 | Efficient Support for NP-based Services | 88 |
| 8.3.2 | Impact of Memory Accesses on Handler Placement | 90 |
| 8.3.3 | ANP-handlers Improve Performance of Data Increasing Services | 91 |
| 8.3.4 | Memory-intensive Services on ANPs | 92 |
| 8.3.5 | XML-based Data Transcoding Stream Handlers Are Feasible on ANPs | 94 |
| 8.4 | Deploying Services on Host-ANP Nodes with SPLITS | 95 |
| 8.4.1 | Efficient Handler Deployment and Configuration | 96 |
| 8.4.2 | CPU Offloading | 96 |
| 8.5 | Importance of Split Services Across Host-ANP Boundary | 97 |
| 8.5.1 | ANP Handlers Reduce Loads Delivered to Host Components | 98 |
| 8.5.2 | Benefits of Offloading Even Non-communication Related Handlers | 99 |
| 8.6 | Summary of Results | 101 |
| CHAPTER 9 RELATED WORK | | 102 |
| 9.1 | Dynamic Service Customization in Streaming Applications | 102 |
| 9.1.1 | Middleware-level Customization | 102 |
| 9.1.2 | Kernel-level Extensions | 103 |
| 9.2 | Modular Frameworks and Service Compositions | 104 |

| | | |
|--|---|------------|
| 9.2.1 | Split Services | 106 |
| 9.3 | Extensible network infrastructures | 106 |
| 9.3.1 | Device-level Research | 106 |
| 9.3.2 | Active Networks | 107 |
| 9.3.3 | Use of Network Processors for Application-specific Services | 107 |
| 9.3.4 | IXP-based Research | 108 |
| 9.4 | Programming Models | 109 |
| CHAPTER 10 CONCLUDING REMARKS | | 111 |
| 10.1 | Contribution | 111 |
| 10.2 | Future Directions | 112 |
| REFERENCES | | 114 |
| VITA | | 123 |

LIST OF TABLES

| | | |
|---------|--|----|
| Table 1 | State variables available to stream handlers at different contexts on the IXP ANP. | 55 |
| Table 2 | Control message exchanged in SPLITS. | 64 |
| Table 3 | SPLITS host-side API. | 68 |
| Table 4 | State maintained by SPLITS runtime. | 69 |
| Table 5 | Effect of handlers on throughput for different sizes of stream data items. | 86 |

LIST OF FIGURES

| | | |
|-----------|--|-----|
| Figure 1 | Intel IXP1200 block diagram. | 10 |
| Figure 2 | Data path through host-ANP nodes. | 12 |
| Figure 3 | Delta Airlines Operational Information System. | 20 |
| Figure 4 | Smart Pointer Application. | 22 |
| Figure 5 | Abstractions used in the programming model. | 26 |
| Figure 6 | Control graph representing a service and its deployment on three processing contexts. | 30 |
| Figure 7 | Activation Points at Rx/Tx and X threads. | 46 |
| Figure 8 | Examples of Rx, X and Tx implementation of a stream handler. | 52 |
| Figure 9 | Position of SPLITS with respect to other layers. | 59 |
| Figure 10 | System components. | 61 |
| Figure 11 | Data path. | 67 |
| Figure 12 | Different data path configurations used in experimental analyses. | 83 |
| Figure 13 | Data delivery delay for IXP- vs. host-level forwarding. | 84 |
| Figure 14 | ANP vs. host throughput for UDP socket stream. | 85 |
| Figure 15 | Client-customized multicast. | 87 |
| Figure 16 | Handler complexity and sustainable throughput for set of services. | 89 |
| Figure 17 | Stream handler performance cost as a function of the amount and type of memory accessed. | 90 |
| Figure 18 | Performance gains for data increasing services. | 91 |
| Figure 19 | Evaluation of an image cropping handler. | 92 |
| Figure 20 | Efficient XML-based data transformation on the IXP. Benefits of parameter selection under increased loads. | 94 |
| Figure 21 | Importance of service deployment across Host-ANP boundaries. | 98 |
| Figure 22 | OpenGL pipeline service. | 100 |

SUMMARY

Modern distributed applications utilize a rich variety of distributed services. Due to the computation-centric notions of modern machines, application-level implementations of these services are problematic for applications requiring high data transfer rates, for reasons that include the inability of modern architectures to efficiently execute computations with communication. Conversely, network-level implementations of services are limited due to the network's inability to interpret application-level data or execute application-level operations on such data. The emergence of programmable network processors capable of high-rate data transfers, with flexible interfaces for external reconfiguration, has created new possibilities for movement of processing into the network infrastructure. This thesis explores the extent to which programmable network processors can be used in conjunction with standard host nodes, to form enhanced computational host-ANP (Attached Network Processor) platforms that can deliver increased efficiency for variety of applications and services.

The main contributions of this research are the creation of SPLITS, a *Software architecture for Programmable LIghtweighT Stream handling*, and its key abstraction *stream handlers*. SPLITS enables the dynamic configuration of data paths through the host-ANP nodes, and the dynamic creation, deployment and reconfiguration of application-level processing applied along these paths. With SPLITS, application-specific services can be dynamically mapped to the host, ANP, or both, to best exploit their joint capabilities. The basic abstraction used by SPLITS to represent instances of application-specific activities are *stream handlers* - parameterizable, lightweight, computation units that operate on data headers as well as application-level content. Experimental results demonstrate performance gains of executing various application-level services on ANPs, and demonstrate the importance of the SPLITS host-ANP nodes to support dynamically reconfigurable services, and to deal with the resource limitations on the ANPs.

CHAPTER 1

INTRODUCTION

Modern distributed applications utilize a rich variety of distributed services. Due to the computation-centric notions of modern machines, application-level implementations of these services are problematic for applications requiring high data transfer rates, for reasons that include the inability of modern architectures to efficiently execute computations with communication. Conversely, network-level implementations of services are limited due to the network's inability to interpret application-level data or execute application-level operations on such data.

This research explores the extent to which programmable network processors, an emergent class of network devices, can be used in conjunction with standard host nodes, to form enhanced computational platforms. The intent is to deliver improved performance and more efficient and flexible service implementations to commercial and scientific distributed applications.

1.1 Background

Data-intensive distributed applications depend upon the performance and availability of underlying services, that range from 'traditional' services like multicast [9] or QoS support [105, 99], to security (e.g., firewalls, encryption [103]), to new classes of services that support XML-based applications [108, 30] or perform data transcoding for multimedia or remote graphics [97, 90, 75, 22, 84]. Since applications evolve over time, and/or end-user interest change, applications require the ability (1) to dynamically place service instances at nodes in the distributed infrastructure, at both data sources and intermediate nodes on the application-level data path, so as to increase the shared resource utilization, and (2) to dynamically configure these services to match the current application requirements (e.g., QoS, fault-tolerance), current end-user interests (e.g., in specific data subsets), or current

operating conditions (e.g., CPU loads and networking resources).

Modern middleware assumes that service placement and configuration occur at the application-level at nodes in the overlay used by the distributed application, where overlays may be as simple as the front-end/back-end distinctions made in large-scale server systems [91, 36], or they may extend across multiple Inter- or Intra-net nodes [6, 101]. Overlay networks [3, 101, 6, 22] have already established the utility of processing stream data ‘in transit’, for media transcoding [36, 90], for sensor data selection and fusion [35], and for handling the large data used in distributed scientific collaboration [109, 110]. Much of the previous middleware- and application-level research [30, 114, 73, 18, 100, 23] has addressed the need of dynamic deployment and customization of application-specific services at user- or even kernel-level at nodes in the overlay [78, 81]. Complimentary work demonstrates the utility of executing compositions of various protocol- vs. application-level actions in different processing contexts [11, 23, 84]. The latter leverage the availability of safe, yet expressive subsets of standard languages for implementation the application-level services, the underlying OS support for dynamic linking and safety checking at the host node, and the availability of the host’s resource-rich environment that can sustain the execution of services of varied degrees of complexity, with arbitrary data access patterns. The costs incurred by all such approaches are the repeated protocol-stack traversals to and from the application layer, the loads imposed at the host node’s CPU and I/O infrastructure, and the overheads of moving large data volumes through a general purpose processing engine (e.g., memory loads).

While application- and middleware-level solutions offer rich functionality on top of standard communication infrastructures, at the other end of the spectrum, research on active networking [106, 62, 92, 5] has demonstrated the utility of extending the core behavior of networks with new functionality, by essentially deploying selected application-specific services into the network. However, in order to address safety and security concerns, and to meet the high data rate requirements imposed at the targeted shared networking devices, this approach imposes restrictions on service complexity and on the data accesses performed. These constraints have resulted in active networking to be better suited for network-level

services, performing operations on protocol headers, such as network monitoring, intrusion detection, routing and multicasting [92, 106, 10, 4], or for simple application services that operate on application-headers, such as those implementing service differentiation, or content-based routing based on request types.

1.2 Motivation

Driven by industry desires to offer new capabilities and services as part of basic network infrastructures [102, 75, 65, 74, 56], device-level research has evaluated the tradeoffs in cost/performance vs. utility of networking devices. As a result, an emerging class of programmable network processors (NPs) is becoming an attractive vehicle for deploying new functionality ‘into’ the network infrastructure, with shorter development times than custom-designed ASICs and with levels of cost/performance exceeding that of purely server-based infrastructures. NP hardware is optimized to efficiently move large volumes of packets between their incoming and outgoing ports, and typically, there is an excess of cycles available on the packets’ fast path through the NP. Such ‘headroom’ has been successfully used to implement network-centric services like software routing, network monitoring, intrusion detection, service differentiation, and others [98, 64, 88, 63, 41].

One outcome of these developments is that increasingly, networking vendors are deploying NPs into their products and are even considering to ‘open’ them to developers to facilitate rapid service development. In addition, industry trends suggest that with the emergence of high-performance system level interconnects, in the near future we can expect tightly integrated programmable NPs as standard components in off-the-shelf systems.

This research aims to better understand how to use programmable network processors. We explore the idea of using NPs closely tied to host nodes, thereby creating a computational platform that can deliver increased efficiency for variety of applications and services. The goals are to attain improvements in end-user application performance, to more efficiently utilize server capacity, and to offer new services at no or little additional performance overheads perceived by end users. Our key idea is to map service functionality to the combined resources offered by hosts and their attached NPs (ANPs).

Such host-ANP pairs are powerful platforms for service execution for several reasons. First, ANP-level service components can benefit from hardware that is optimized for the efficient manipulation of ‘in transit’ data, as it is being streamed into and out of the ANP. Second, ANP-level service execution can reduce hosts’ memory or CPU loads, particularly when their actions involve data forwarding (e.g., proxy forwarding) or replication (e.g., multicast). Next, host-ANPs offer high degree of flexibility and efficient reconfigurability, which are required by the targeted distributed applications. Finally, since ANPs’ capabilities are limited, a software architecture supporting the execution of service components on both ANPs and on the hosts to which they are attached further enhances a developer’s ability to efficiently implement end user services. Examples of the combined use of ANP- and host-level service functionality abound, ranging from earlier work that statically maps selected portions of protocol stacks onto communication co-processors, or that implements synchronization or transactional primitives there, to more recent work on dynamically extensible communication machines [34, 88, 56].

1.3 Thesis Statement

- Application-specific processing is not only feasible on network processors, but certain types of application-specific functionality can be implemented more efficiently than on standard host nodes.
- Limited NP resources are not sufficient for efficiently sustaining services of arbitrary complexity.
- Joint use of hosts with attached programmable NPs, creates platforms that are better suited for the types of application-specific functionality commonly required in today’s distributed streaming applications.

1.4 SPLITS and Stream Handlers

The main contributions of this research are the creation of SPLITS, a *Software architecture for Programmable LIghtweighT Stream handling*, and its key abstraction *stream handlers*. SPLITS enables the dynamic configuration of data paths through the host-ANP nodes,

and the dynamic creation, deployment and reconfiguration of application-level processing applied along these paths. With SPLITS, application-specific services can be dynamically mapped to the host, ANP, or both, to best exploit their joint capabilities. The processing contexts on the host and the ANP are treated as nodes in a micro-overlay, where arbitrary data paths can be formed so as to best utilize the resources and the functionality available at each context. The basic abstraction used by SPLITS are *stream handlers* - parameterizable, lightweight, computation units that operate on data headers as well as application-level content. Application-level services composed with stream handlers can run on the ANP, the host, or across host-ANP boundaries. To enable the application-level processing performed by stream handlers, SPLITS provides ANP-level runtime support for message assembly and fragmentation, for dynamically deploying stream handlers onto ANPs, for configuring the data path through the host-ANP contexts, and for configuring stream handlers while in use.

In contrast to earlier work on extensible communication co-processors used in cluster machines [88, 34, 102], SPLITS is targeted at the services provided in overlay networks or in ‘application servers’ used by service providers. The classes of services addressed by SPLITS include:

- transactional services, like the data mirroring performed in the operational information systems used by large corporations [43, 70, 42];
- data translation services, to help applications deal with multiple data formats, alternative data representations, and the up/down translations they require [108];
- media services that can efficiently perform tasks ranging from data transport and transcoding [75, 36, 90] to data personalization on behalf of individual end users, as in scientific collaboration or remote graphics [109]; and
- event services, like those used in wide area notification or sensor nets [113, 89, 44].

The performance gains attained from using SPLITS for implementing application-level services stem from (1) the ability to use optimized, highly parallel NP hardware with built in support for tasks like queuing, scheduling, and signaling, (2) the removal of load from host CPUs, (3) the offloading of the host’s networking and I/O infrastructure, and (4) the

joint use of host and ANP resources.

SPLITS and stream handlers are implemented for hosts that run standard Linux OS kernels and for ANPs that are based on Intel's IXP network processor. Services realized with SPLITS include (1) application-specific data mirroring as used in enterprise applications like operational information systems, (2) the efficient processing of XML-structured data, and (3) remote graphics and visualization actions that perform per-client customizations of visual displays.

SPLITS occupies a 'middle ground' in the spectrum of work described above. For overlay networks, we demonstrate that performance can be enhanced by judiciously mapping overlay functionality onto network processors vs. the hosts to which they are attached. This is particularly evident for overlay operations that cause data replication, as when an overlay node mirrors data to multiple remote nodes. Measurements in Section 8.2 demonstrate substantially improved end to end latency when doing content-based forwarding of data streams on ANPs vs. hosts. Such improvements are not surprising: ANPs are designed for efficient data streaming whereas hosts are not. Performance improvements are most pronounced when ANPs are used to enhance host-level operations. ANP-level stream handlers that filter the data streamed to and operated on by hosts, for example, reduce the loads imposed on hosts' I/O busses and memory structures and decrease the workloads imposed on host CPUs. Application-specific ANP-level data filtering and selection are shown to improve the performance of data mirroring with the data streams used in an Operational Information System by 25% (see Section 8.3.5). Such improvements are particularly striking when ANPs operate on efficient, binary representations of structured data, in comparison to the inefficient XML-based data representations currently used by many web services applications and infrastructure. Finally, even mid-range ANPs like Intel's IXP1200 are surprisingly powerful. Results shown in Section 8.3.4 show that this ANP is capable of performing client-specific data reductions even for large data events and when a high percentage of the events' payload must be inspected, the example being dynamic image cropping for OpenGL-based remote graphics displays performed at gigabit speeds. In general, of course, there is a near-linear relationship between the ANP's ability to offer

high throughput and the percentage of message payload touched by it [41]. This is a key reason for our development of the SPLITS software architecture that permits services to be composed of stream handlers deployed across the host-ANP boundary.

Superior performance of host-ANP pairs compared to non-programmable host/network card infrastructures is not surprising. Essentially, stream handlers on ANPs can be written to enable hosts to focus on the computationally intensive application processing they do best, while ANPs perform the ‘fast path’ packet processing tasks for which they are well-suited. For example, ANPs are designed to efficiently perform data replication tasks like multicasting, whether such actions are taken based on protocol- or on application-level headers. In contrast, such ‘data increasing’ tasks executed in host-resident overlay networks require hosts to repeatedly execute protocol stacks, may imply multiple crossings of protection boundaries, and can result in unpredictable performance levels. These latter two problems have been well-documented, giving rise to kernel-level solutions to TCP-tunneling or proxy forwarding [87], for example. SPLITS addresses these issues by handling stream processing with the host vs. ANP resources that are best suited for certain tasks.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 describes the pair of host and attached network processors, and motivates their joint use. Chapter 3 discusses the targeted domain of streaming distributed applications, and gives more detail on two applications in whose context the thesis work is evaluated. Chapter 4 presents the programming model used by application developers to map computations across the joint host-ANP resources. The stream handlers, their properties, and their use for representing application-level services, are explained in greater detail in Chapter 5. The software architecture SPLITS, its components, the manner in which it enables dynamic deployment and configuration of stream handlers and data paths through the host-ANP nodes, and current implementation status are described in Chapters 6 and 7. The experimental evaluation of the ideas presented in this thesis appears in Chapter 8. A more detailed survey of related work along several different dimensions is given in Chapter 9. Finally, Chapter 10 highlights

open questions for future research, and summarizes our conclusions.

CHAPTER 2

HOST - ATTACHED NETWORK PROCESSOR PAIRS

This chapter motivates in greater detail the coupled use of standard host systems with attached network processors. It presents a high-level view of the host-ANP platform and the data path through it. We highlight the main features of network processors, in order to support our choice of using NPs to enhance hosts' capabilities, and briefly describe the Intel IXP network processors used as a sample NP.

2.1 Network Processors

Technology advances have created programmable network processors that can be used across the entire range of moderate data rate embedded systems like residential gateways [65] to high end data streaming at 10GB link speeds [50]. A concrete example is Intel's IXP line of network processors. At the low end, the IXP 425 board targets embedded systems applications like residential gateways and in fact, is already used in one of Linksys' new products [65]. The Intel IXP1200 used in this research is a mid-level product, and its commercial use has included Video over IP [75] applications. At the high end, the IXP2800 and IXP2850 operate at gigabit speeds, with new developments targeting the 10GB market [25].

There is a large variety of NP products on the market, with 40+ vendors competing for their market share [107, 94]. The objective is to deliver to service providers, network administrators, and others, greater flexibility and shorter deployment time of new architectures, services, or protocols into the network, than ASIC-based solutions. They all share some similarities. Programmable network processors are based on on-chip multiprocessing processing elements (PEs) with varying complexity, ranging from many, designed for a dedicated task, as in EZchip's or Vitesse's approaches [31, 95], to few and complex as in the Intel's IXPs or Motorola's C-Port NPs [50, 49]. In addition to the PEs, most NPs typically include additional hardware support for network-related functionality, such as switch fabric

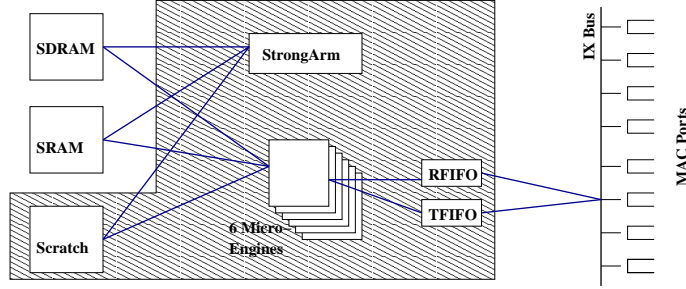


Figure 1: Intel IXP1200 block diagram.

management, look-up table management and queue management, different types and sizes of on-chip memory and controllers for additional memory, a special processing unit to handle control and supervision, proprietary internal bus, etc. Some higher-end NP products offer additional hardware support such as for packet buffer management or crypto functionality [49, 32]. The NPs are primarily targeting flexible development of network-level functionality and efficient operation in high-data rate environments. Our work exploits the use of NP resources for implementing application-level functionality. Due to resource limitations on the ANP, we consider more specifically, the joint use of NPs attached to standard hosts. The goal is to benefit from NPs' optimized hardware for communication-related tasks in the application.

2.1.1 Intel IXP Network Processors

Intel IXP1200 Network Interconnect Boards. High cost/performance for data streaming in network processors like the IXP1200 is attained by use of parallelism in stream processing. The Intel IXP1200 [50] is a multiprocessor on a chip containing a StrongArm (SA) core and six microengines with four thread contexts each (see Figure 1), all operating at 232MHz. Each microengine has a 2Kword instruction store loaded by the SA and shared among the four contexts. The chip integrates memory controllers for up to 256MB of SDRAM, 8MB of external SRAM and 4KB of internal scratch memory. Ethernet or other MACs connect externally through a proprietary 64-bit/66-MHz IX bus accessed through on-chip transmit and receive FIFOs, and network packets are received as a sequence of 64-byte MAC-layer packets. A 32bit/66MHz PCI bridge links the IXP1200 to the host or

to other boards capable of PCI-based interactions (e.g., FPGA boards used for applications like intrusion detection). Our work targets the IXP1200 chip on Radisys ENP2505 boards, which have the maximum amount of memory and four 100T Ethernet MACs.

Intel IXP2xxx family. We are currently migrating to the next generation IXP2400 network processor, which offers an increased number of microengines at higher speeds, additional memory, a faster host-ANP PCI-based interconnect, and other technology improvements. The IXP2400 chip [53] includes eight 8-way multithreaded microengines for data movement and processing, local SRAM and DRAM controllers and an integral PCI interface with three DMA channels. The Radisys ENP2611 board [83] on which the IXP2400 resides includes a 600MHz IXP2400, 256MB DRAM, 8MB SRAM, a POS-PHY Level 3 FPGA which connects to 3 Gigabit interfaces and a PCI interface. An XScale core, running Linux, is primarily used for initialization, management and debugging. Our design uses host-side drivers with NP firmware to implement the host-ANP interface. The IXP2400 is attached to hosts running standard Linux kernels over a PCI interface.

2.2 Host-ANP pairs: High Level View

Host-ANP pairs are tightly coupled platforms consisting of standard hosts connected to a programmable network processor via a system-level interconnect. Our work uses network processors attached to the host via its PCI interface. In this configuration, the host can use the NP as a programmable network interface card. Using a network processor as a ‘smart’ network interface can improve system performance by offloading the host processor and by reacting to important events with lower latency. Essentially, the NP adds to the network interface the ability to aggregate, classify and reorganize data as it is being transferred [68]. The host and the processing contexts on the network processors (e.g., threads running in separate microengines on the IXP1200 NP), share the joint host and NP resources, and jointly contribute to data path processing. Externally, a host-ANP pair represents a single processing platform, which communicates via the network interfaces available on the host or on the attached NP. Internally, these platforms represent distributed system consisting of host- and ANP-level processing contexts, that share the joint host-ANP resources, and

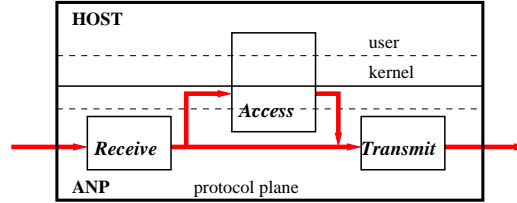


Figure 2: Data path through host-ANP nodes.

communicate via controlled access to shared memory channels. Some, or all of these contexts can be involved in the processing along the data path, thereby forming micro overlays.

While these platforms can be used in a standard manner, with computation-related tasks confined to the host, our focus is on the potential of jointly using the host and ANP for computational data paths that cross host-ANP boundaries or that are fully ANP-bound.

2.2.1 Data Path Through the Host-ANP Node

An application’s data path can traverse host-ANP resources in several ways. First, data that originates at the host, or is delivered to the host on one of its NIs, can be transmitted through the ANP’s interfaces, i.e. ports. Protocol processing can be performed on the outgoing data, with ANP-resident processing engines partitioning the data into packets and adding the appropriate headers (i.e., splitting the protocol stack across the host/NIC boundaries [13, 12, 88]). Examples of other useful actions executed on the outgoing data stream include mirroring, multicast customized on per-destination basis, stream differentiation, etc. Next, data delivered on one of the ANP’s ports can be placed into host memory, either directly via DMA, or after the appropriate pre-processing is performed on the ANP. Sample actions include performing protocol processing to merely assemble the application-level data and place it into memory, real-time packet scheduling, and filtering actions on the incoming data. Finally, the data path can be fully confined to the ANP, as is the case with host-ANP pairs used as intermediate nodes in application-level overlays. The functionality implemented on the ANP includes basic data forwarding to its ultimate destination in the application-level overlay, or it can be enriched with additional processing. The intent, of course, is to deliver exactly and only the data currently needed by each node, using application-level information about message formats, data layout and content.

Figure 2 represents the basic elements of the data path through the host-ANP pair. The *Receive* block on the ANP executes the receive-side protocol code to detect message and flow boundaries network across packets, and it associates packets with the corresponding memory queues. On the IXP1200 NP used in our work, this functionality is executed by a set of Rx threads, one per incoming port, all assigned to a single microengine. Received data can be forwarded directly to the transmit-side protocol code – *Transmit*, which directs it to its destination. Our implementation assigns four Tx threads, all from the same microengine, to execute the transmit-side functionality. Additional processing performed between packet receive and transmit represents the functionality implemented by the *Access* block. Such processing is implemented with additional application-specific codes, which are executed by one of the ANP-level processing contexts (termed X contexts), in the host’s OS kernel, or at user-level. Our work further enables developers to embed such *Access* functionality in the *Receive* or *Transmit* contexts.

CHAPTER 3

APPLICATION DOMAIN

This chapter presents the applications considered by our work, which are large-scale, distributed, data-intensive, streaming applications, that use dynamically configurable services. The goal is to demonstrate that our work considers a significant target application space and that some of the specific requirements posed by these applications can be implemented efficiently on host-ANP nodes.

3.1 Streaming Applications

This research targets data-intensive distributed streaming applications, which deal with both large data volumes and large data sizes. In general, in these applications, structured data is transmitted on a continuous basis, often relying on application-level overlays to direct the data stream(s) from sources to the destination set. Furthermore, these applications depend upon the ability of the underlying infrastructure to provide services for online data analysis, for pre-processing and/or customization of a stream before it reaches its destination, or for stream manipulations necessary for the implementation of different quality or fault-tolerance properties. The stream processing actions involved in these service implementations are executed at the stream end points, sources and destinations, as well as at intermediate nodes in the application-level overlays. Such actions are not only application-specific, but they also require access to, interpretation, and manipulation of application-level content of the messages traversing the overlay.

There is an abundance of classes of streaming applications, distinguished by their large data needs and by the substantial processing that is needed to convert raw data into information useful to end users. Some of the classes of applications relevant to our work include:

- *Scientific Collaborations* – in remote scientific collaboration [77, 109], distributed

instruments, remote sensors, and visual displays interact in real-time with human end users and with large-scale scientific or engineering simulations, to perform industrial tasks like collaborative parts design, to improve scientists' understanding of certain physical processes (e.g., consider the ongoing SuperNova Simulation Initiative at DOE), or to deliver real-time services to end users, as in weather prediction. These applications depend upon the availability of services that will provide support for efficient sharing of information among multiple users and application components and enable customized delivery of data to the specific destination sets based on current operating conditions, data content, and application-specific quality requirements [109, 37, 73]. These application also rely on data translation services, to help applications deal with multiple data formats, alternative data representations, and the up/down translations they require [108].

- *Operational Information Systems* – in Operational Information Systems (OIS) used by large corporations, such as Delta Airlines, FedEx, and Worldspan, data captured by sensors or entered by human operators is transformed into the information needed to run the company's daily operations [43, 70]. These applications require services that will gather, transport, and transform data that is exchanged among the company's subsystems, while also executing business rules to extract useful information from the data streams and perform the necessary system state updates. The critical nature of the operations executed by the OISs poses replication requirements, which in turn translate in the need for services that will efficiently perform transactional tasks, such as mirroring of streaming data, or delivery of subsets of data to specific subsystems. Furthermore, company internal optimized subsystems interact and must share information with external clients and applications, and therefore require the ability to efficiently perform necessary data format translations.
- *Multimedia Applications* – multimedia applications such as remote visualization or dynamic delivery of camera-captured data require media services that can efficiently

perform tasks ranging from data transport and transcoding [75, 36, 90] to data personalization on behalf of individual end users, as in scientific collaboration or remote graphics [109]. These services can be specified with application-specific quality metrics that requires dynamic functionality such as downsampling, transcoding, scheduling, and cropping [116, 58, 78, 90]. Furthermore, the specific implementation of the required functionality is dependent upon the specific media encoding used and the algorithms used. Media services are particularly useful in wired-to-wireless gateways that deal with limitations in available wireless bandwidths by reducing the data volumes sent to clients [65].

- *Event Notification Systems* – event notification systems such as those used for stock ticker updates or monitoring of traffic conditions [21, 113], sensor applications [61, 35], or applications supporting dynamic web content delivery [66, 16] typically use publish/subscribe infrastructures, where boolean or simple query-based selection operations and routing actions applied to events ensure that only necessary data items are transmitted at intermediate points of their overlays [113, 89]. These application require ability to efficiently determine subsets of events relevant to a specific client or class of clients, and replicate that subset on all of the corresponding connections.

3.2 Network Processors and Application-specific Services

Common to all of these applications are the requirements for services that enable efficient movement of large data volumes, perform application-specific stream manipulations, and deal with the dynamic nature of the application’s inputs, component interactions, outputs, and current QoS needs. Such dynamics, coupled with runtime variations in network conditions and system resources, imply that QoS needs cannot be met without runtime modifications to applications’ and data streaming functionality [79].

This thesis shows that the host-ANP pairs introduced by our work are particularly well-suited for such streaming applications. The NPs’ ability to efficiently move large data volumes improve the performance of the data transfers required by streaming applications. In addition, the excess of cycles associated with data movements can be used to implement

some of the application-specific services required by applications, or to enhance the host's ability to execute them, by removing loads from the host's CPU and I/O and memory infrastructures. Finally, the programmability of NPs can be used to deliver the dynamically reconfigurable platforms necessary for addressing changes in application interests, QoS specifications, or operating conditions. Processing actions executed on the ANP can implement a wide range of stream manipulations to support the dynamic needs of applications, and can be categorized as follows:

- *content-based routing* – necessary for content-based load balancing [7], or for wide area event systems, like IBM's Gryphon system [113], where ANPs perform the boolean or simple query-based selection operations performed on events, to route events or to deliver suitable event subsets to groups of end users [89];
- *data sharing* – required for data mirroring and replication services, used in publish/subscribe infrastructures and transactional services such as those executed in OISs;
- *selective data filtering* and *data reduction* – needed to ensure delivery of only those data items and those elements of items that are of current interest to the application, as with image cropping that matches the current end user's viewpoint in remote scientific collaborations, or simply reduces the amount of data placed on the network so as to match the current networking conditions;
- *data transcoding* operations, used in OISs or scientific collaborations, for instance, to enable data exchanges that have the required format or contain the necessary level of detail;
- *priority scheduling* necessary to implement QoS-centric stream manipulations, e.g., to order or classify the data items being transported as in for critical data delivery in multimedia or sensor applications [75, 116];
- ancillary tasks like data stream *monitoring* and *control* used for timely detection of critical events in sensor applications, or for status services such as online averages and sums; and

- *security* services that cover the wide range of security processing either already embedded into high end networking products (e.g., the crypto processing in the IXP2850) or under consideration for such machines (e.g., authentication, intrusion detection, attack forensics, etc. [26, 71, 24]).

We cannot list all possible communication services host-ANP pairs are capable of performing. Instead, we have outlined some of the basic services that can be used to implement the potentially high payoff functionality currently considered by vendors and/or researchers.

3.3 Sample Applications Using Host-ANP Pairs

Next, we present in greater detail the two sample streaming applications most extensively used in our work.

3.3.1 Operational Information Systems: Delta Airlines

Our first focus is on applications that require cluster-based server solutions for their continuous data streams, an example being the Delta Air Lines OIS described in more detail in [43]. An OIS is a large-scale, distributed system that provides continuous support for a company's or organization's daily operations. OISs are used 24/7 in companies like FedEx, UPS, or Delta Air Lines [43, 70], to collect, transport, collate, and distribute the information used in these companies' daily operations. One example of such a system we have been studying is the OIS run by Delta Air Lines, which provides the company with up-to-date information about all of its flight operations, including data events about passenger boarding, flight arrivals and departures, flight positions, and baggage. Information is collated using business rules, which in turn generate events that are used for tasks ranging from the update of airport terminal displays to notifications sent to caterers of passengers' food preferences.

Multiple cluster nodes apply business logic to continuous input streams comprised of FAA flight position updates and Delta-specific flight information (see Figure 3). The cluster nodes form an Internal Event System (IES), that interacts with clients, by generating continuously derived system-state updates and/or responses to explicit requests. Different

nodes may be dedicated to different OIS' subsystems, such as the Flight Progress Event System (FPES) that maintains all relevant flight information, or a separate subsystem that is responsible for all interactions with the passengers database. In addition, subsystems may be replicated across multiple nodes, in order to meet the system's availability and reliability requirements.

The large number of clients, the complexity of the business logic being applied and its working set size of hundreds of gigabytes, and a 24/7 uptime requirement dictate a cluster-based solution to the stream processing done by a business server of this nature. Other examples of such cluster-based streaming servers include sensor processing [45], graphics and visualization servers [77], transaction processing engines, and 'fresh information' database servers [82].

Traditionally, in OISs, dedicated machines serve as the cluster front-end, termed 'application server', and perform actions like format translations, load-balancing, data striping, or simply serve as a software router. ANPs enhance such dedicated nodes. In the Delta OIS example, streams entering the cluster-server are pre-processed by the NP's processing engines, and based on application-level rules, they are mirrored to the desired destination node(s). Other actions useful for applications like these include stream differentiation based on data content or stream customization based on destination nodes, filtering, and online derivation of new data such as averages and sums. Dynamic changes in certain parameters of the processing can result in modified performance, and such changes can be used to dynamically tune the performance of the system to current operating conditions.

An example is the use of host-ANP pairs to offer efficient solutions for the data sharing tasks of large-memory applications like FPES. Specifically, by using techniques like *selective or adaptive data mirroring* [43] on the NPs at the ingress points of FPES, duplicate server systems can be provided with the data necessary to handle the state refresh tasks required in such situations. Duplicate servers need not execute all of the tasks run by the primary FPES, but their state and actions can be constrained to focus on the specifically needed data. In turn, data mirroring does not simply duplicate all data received by FPES, which would lead to unnecessary network and server loads, but instead, data can be selectively

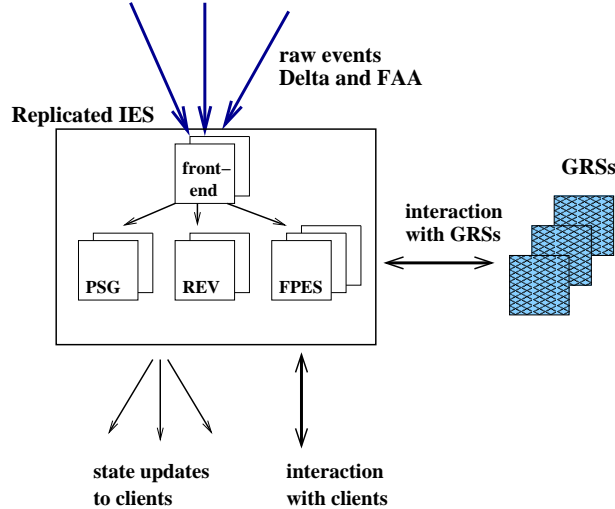


Figure 3: Delta Airlines Operational Information System.

mirrored to focus on what is needed for emergency response.

Data mirroring at ingress points is one ANP-level service of use to applications like FPES. Another useful service is *selective data forwarding* at egress points, where data events produced by FPES are routed to appropriate other subsystems, such as the General Reference System (GRS) used for interactions with external service providers, e.g., caterers. Again, unnecessarily high server loads or network usage can be avoided by selecting at the egress ANPs exactly the data needed for such actions. Furthermore, removal of some of the content produced by the FPES may be required to meet the company’s privacy policy and confidentiality requirements. Finally, the data translations at FPES egress are necessary in order to convert it to a representation that can be accepted by the external subsystem, and can be efficiently implemented on ANPs.

In general, the tasks listed above require ANPs to use information about application-level data structures and layouts in message payloads. In in-house systems, such information is readily available as descriptions of the event formats used. More general descriptions of such formats in efficient binary forms are described in [108, 17]. This raises a third issue, which gives rise to another class of important services ANPs may execute on behalf of applications: the dynamic management and use of meta-information about data structure.

The approach to efficiently representing and using information about data formats used

in our work leverages the widely used XML standard. XML schemas are used to represent meta-information, used and manipulated by hosts. ANPs do not utilize these inefficient descriptions, however. Instead, schemas are dynamically translated to efficient binary formats [108] and ANPs use dynamically deployed binary formats to understand and then manipulate the payload of application messages. In this fashion, ANPs can efficiently perform ‘XML’ services like data selection, data subsetting, data reorganization, and similar changes to data representation. ‘XML’ services on ANPs are evaluated in Section 8.3.5.

3.3.2 Scientific Collaborations: The SmartPointer Framework

A scientific application driving our work is a framework for collaboration across physicists, chemists, and application researchers (e.g., Aerospace). This application, termed ‘SmartPointer’ due to its realization of multiple views and ‘pointers’ into the large data space shared across researchers, is depicted in Figure 4 and discussed in detail in [109]. Here, we focus on those parts of the application that would most benefit from embedding functionality in the ANPs attached to the cluster server transforming its data.

The central pieces of a molecular dynamics application built with this framework are a *bond server* and *imaging servers*. The bond server receives streams of molecular data, from a running simulation or from stored earlier runs, and computes the different types of existing atomic bonds. It then forwards the newly created information to imaging server(s), which prepare easily rendered representations of this information. Different classes of display/manipulation devices interact with the imaging server and request all, or parts of the molecular dynamics data. Differences in quality of service requirements exist across classes and within classes, across multiple clients. There can be multiple imaging servers, each perhaps driving a certain class of clients, as with an imaging server that executes the entire OpenGL pipeline on behalf of its low-end, handheld clients, compared to an imaging server that relies on OpenGL or even higher level display capabilities (e.g., CAVE software [86]) to exist on its clients.

The figure assumes that the middleware used to ‘connect’ information sources, transformers, and sinks uses the publish/subscribe model of communication. In this model, the

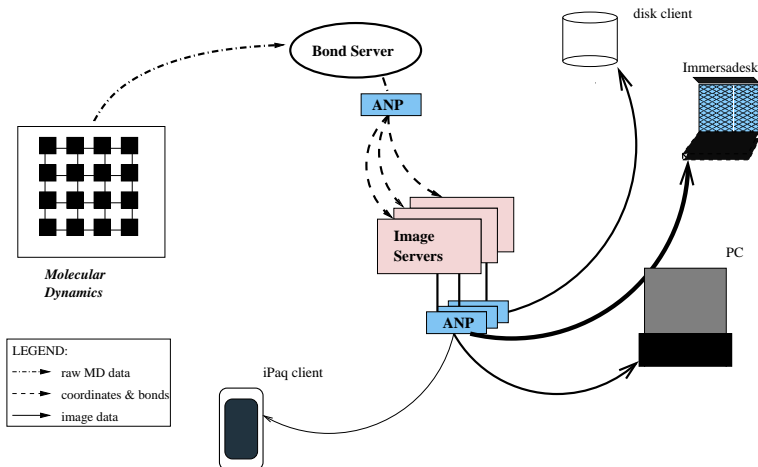


Figure 4: Smart Pointer Application.

data units sent are events, and event distribution uses logical communication channels (i.e., event channels) to which information sources and sinks subscribe whenever needed. QoS requirements translate in different actions being applied to events when they are emitted by source, accepted by a sink, or both. By default, the ECho event system [30] used in our research maps all source-to-sink communications to reliable point-to-point network connections. If desired, however, developers can also map event handlers into ‘third parties’, which is useful when dynamically mapping the event handlers that perform computationally expensive data transformations into non-source or non-sink servers like the Terastream server [109]. Such mappings result in the creation of overlay networks.

We are not concerned with middleware details. Instead, our focus is on the mapping of application-specific services to either application-level processes or ‘into’ the underlying interactive grid platforms, consisting of host-ANP nodes. First, an ANP placed on the data path ‘between’ the bond and imaging servers can customize the molecular data stream for a certain class of imaging servers, as exemplified by an image server for low end clients which cannot display high resolution data in the first place. Second, an ANP placed ‘between’ imaging servers and remote clients can support the per client, dynamic stream customization (e.g., image cropping or downsampling) necessitated by dynamic bandwidth constraints in Internet-connected clients or by power constraints in clients running on handheld devices. In all such cases, the functions executed by such ANPs manipulate the application-level

content of data events. Finally, data selectivity is also important when client requirements change, as with a client that frequently changes its data focus.

For the SmartPointer application, we demonstrate the utility of host-ANP pairs for executing selective data filtering that uses application-specific knowledge and data content to determine the subsets of data currently required by the imaging server. In addition, we show that ANPs can help with the dynamic personalization of remote graphics services. Specifically, the imaging server runs an OpenGL graphics pipeline, which uses the host's graphics processor to perform computationally expensive operations like perspective transformation. Such a fast graphics host is capable of producing high quality graphical displays for multiple remote clients. Clients, however, may require image personalization, ranging from data downsampling that reduces the color depth of displays (e.g., to reduce energy usage) to image cropping, which permits each client to focus on those portions of the display currently important to it. Specifically, Section 8.3.4 demonstrates experimentally that even Intel's mid-range IXP1200 ANP can execute personalization operations like image cropping at Gigabit speeds. As a result, it is conceivable for an ANP to take a single graphics stream received from its host and personalize it for a number of clients. Furthermore, the actual personalizations being performed are easily parameterized, as with a client providing current view area descriptions to the host-ANP pair, which are then used to control the ANP's cropping actions. This could open up interesting network-level opportunities for future remote graphics and gaming services.

CHAPTER 4

PROGRAMMING MODEL

We next present the programming model upon which we base the development of the proposed SPLITS software architecture. This chapter outlines the basic abstractions used in our model, the interactions among them, and compares this model with existing programming models used with programmable network infrastructures or developed for streaming applications.

4.1 Model Overview

SPLITS enables the dynamic deployment and configuration of application-specific services on the data path through the host-ANP nodes. The intent is to support the exchanges of large data between the different components of distributed applications. Such an exchange is modeled as *data stream* from data source to sink. A *path* is the sequence of physical or logical processing contexts, interconnected via communication channels, that are traversed by each data item in the stream as it is delivered from source to sink, and it may include other execution contexts that exist in the distributed system. The default functionality executed by these intermediate contexts involves the forwarding of the data items to the next context on the path towards the stream's sink: data is received, the next context is determined, and it is transmitted on the path's outgoing links at this context.

The computation applied to an application-level message in the stream represents a *service*. A service can operate on each message individually, or it can implement some property across multiple messages. Each service can be decomposed into multiple basic components termed *activities* that may be executed by different execution contexts.

The SPLITS model is aimed at distributed streaming applications, and it can be used by programmers to represent simple and complex network- or application-level services.

4.2 Basic Abstractions

The basic abstractions used in the SPLITS model, are summarized in Figure 5. Their discussion is illustrated with examples drawn from the Delta Airlines application described in the previous chapter. In this application, a data stream is comprised of business events exchanged among different components of the airline’s distributed system. One instance of such a data stream is the set of events sent from the Delta General Reference System (GRS) (see Figure 3) to an external caterer. The path traversed by this stream includes the source GRS, the caterer’s receiving context, i.e. sink, and the intermediate network- and application-level routers that forward data to it. Sample services executed on this path include (1) filtering applied to the event stream, so that only events of interest (e.g., events that carry information for a specific flight or airport) are delivered to the caterer, and (2) format translation applied to each event, so that its representation can be interpreted by the caterer.

Data abstraction. The basic data abstraction in the SPLITS model is a *self-describing application-level data unit*. A data unit is self-describing through information embedded in the data unit’s header, which can specify the data size and format, i.e. layout, offsets, and sizes of the application-level data fields that comprise the data unit, as well as the data unit’s membership and position in a set of similar data units from the same source(s) or for the same destination(s). This information defines a *data tag* with two parts: one that defines the application-level data represented by the data unit, the other that determines its membership in a group, based on network- and/or middleware-level protocols. In the Delta Airlines application, data units correspond to Delta’s business events - internal Delta messages that are exchanged between different components of the Delta OIS. Each Delta event has a binary representation that carries information about the event’s type in the form of a format identifier. The format identifier, along with lower-level protocol headers, make up the data tag.

A data unit may be represented as a sequence of one or more *data fragments*, all of which, except for the last one, are of size *fragment size*. Each fragment is tagged with a *fragment tag* that uniquely binds a fragment to a single data unit, and to a specific position

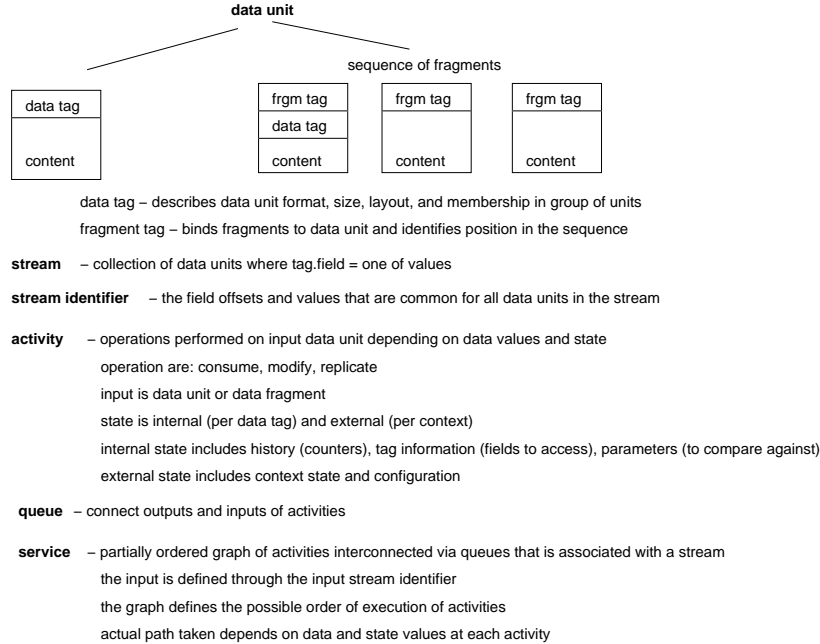


Figure 5: Abstractions used in the programming model.

in the sequence of fragments that compose the data unit. The first fragment in the sequence is the only one that has to contain the data tag corresponding to the application-level data unit.

A single application-level data unit can have multiple representations as a sequence of fragments. For instance, an application-level data unit can be represented through a series of memory buffers that contain portions of it. Each buffer has an identifier binding it to the specific application-level data it represents, and identifying the buffer’s position in the sequence. The same application-level data unit can be represented as a sequence of network packets. The header information in each packet uniquely identifies the application-level message to which the packet belongs, as well as its sequence number. In this case, the fragment can be an IP packet, an Ethernet frame, or one the MAC-layer packets of which the Ethernet frame is composed, for instance. In both representations, the data tag is contained only in the first fragment. All subsequent fragments are tied to it through their fragment tags.

Computational abstraction. We refer to the basic computational unit used in our model as an *activity*. An activity has one *input* - the data unit to which it is applied, and one

or more *outputs*. An activity can perform one or more elementary operations on a data unit: consume, modify, or replicate a data unit. Whether or not a data item is consumed or replicated, or how it is modified depends on its content, as well as the activity's *internal* and *external state*. Each activity can access and manipulate its internal state, which includes historic information (e.g., counters), information about the data unit currently being processed, such as data tag, size, offsets of fields that need to be accessed, and parameters that can further specify the exact operations to be performed. In the Delta example, an activity filters out the Delta events that are not relevant for a specific flight. Here, the internal state represents the fragment counters necessary to access the specific offset of the data field that corresponds to the flight number and the specific flight number of interest to the application. The actions taken are either to consume the event or to replicate it to a single output. External state is a collection of system-wide properties that may be accessed and manipulated by more than one activity or by other application components. These include information about the processing context in which the application executes, the global state that defines the execution of the underlying communication protocols, etc.

Some activities can be associated with more than one data tag. In such cases, a subset of internal state may be required for each data tag, in order to specify any differences in the operations that need to be performed. For instance, an activity that reduces the update frequency by consuming every *n*th data item can be used on Delta events that update terminal displays, or on FAA flight position updates. However, while it may be allowed to halve the frequency of display updates, FAA regulations may specify that only one in ten FAA position updates can be ignored. The internal state is then used to maintain the counters and the parameters for both data types independently.

An activity is *incremental* with respect to a data fragment if executing it on the entire application data unit results in the same outcome as when executing it on each fragment in the sequence separately. Such is the case with activities which, when determining which operation to perform, and how to perform it, only require information contained in the current fragment. Furthermore, an activity can be incremental with respect to multiple fragment sizes. For instance, IP forwarding is incremental with respect to a 64B MAC layer

packet, since all information required to determine the next hop address is contained only in this fragment. Similarly, it is also incremental with respect to a fragment that corresponds to an Ethernet frame. The Delta filtering activity is incremental with respect to a 64B MAC-layer packet as well, since the offset of the flight number field is such that this field fits within a single 64B fragment. The Delta format translation activity is not incremental with respect to any fragment size, since it needs access to different fields of the original Delta event, whose layout is neither consecutive, nor aligned on some constant offset.

Given the above, the properties of an activity include:

- the fragment sizes with respect to which it is incremental;
- the input data items with which it can be associated, whose type and sizes are defined through the data tags;
- the operations performed on the input data item;
- the number of outputs, and data tags (and sizes) associated with each output (as a function of the input data tag);
- the internal state it accesses and manipulates for each of the data tags; and
- the subset of external state it requires.

Communication abstraction. Activities interact with one another through shared *queues*, which connect their inputs and outputs. A queue is a communication abstraction, similar to channels or pipes found in other programming models [106, 51]. Two primitives *enqueue* and *dequeue* are used to deliver data to and from the queues. Multiple activities can act as sources or sinks for the same queue.

Each queue is defined by the data units it can contain, its sinks, sources, and its capacity. The queues in our model are a communication abstraction only, we do not specify whether there should be data copying involved with the queue operations, or whether data can be exchanged by reference only. A queue can be an array of memory buffers containing application-level messages or pointers to messages, or it can be the sequence of packets delivered on an incoming/to an outgoing connection, etc.

4.2.1 Data Streams and Services

Data stream. The collection of data items for which specific fields in the data tag have specific values, or sets of values, define a *data stream*. The fields in the tag that unify the data items in a stream can be solely the application-level portion of the data tag, such as format identifier, or information about the path endpoints, such as source and sink addresses, or a combination of both. The subset of the data tag shared by all items in a stream represents a *stream identifier*.

For instance, in the Delta OIS, all FAA data events share the same format identifier ‘FAA’. A stream of all FAA events is then the collection of all data units where the format field in the data tag matches the value ‘FAA’. A stream of all data from a set of sources is the collection of all data items where the data source field in the data tag matches one of the source addresses.

Services. The collection of activities applied to data units with a certain tag, and the queues that connect them, define a *service*. A service can be represented as a partially ordered graph of activities (see top part of Figure 6). Each path through the graph represents a possible order of execution of the activities. The specific path taken depends upon the exact operations executed by each activity, which depend on the data tag, its content, and the external and internal state.

The activities that compose the service can be deployed and executed at one or multiple processing contexts. We next define how the deployment of services is supported by this model.

4.3 Activation Points

The data path through a context involves the following operations: data is delivered to the context, its destination is determined, and data is delivered from the destination. A data destination is always a queue. The queues that connect activities in different contexts are multiplexed on top of the communication channels that connect these contexts, such as shared memory queues, for instance. Within each context, queues connect activities mapped to well-defined activation points. *Activation points* are locations in contexts where

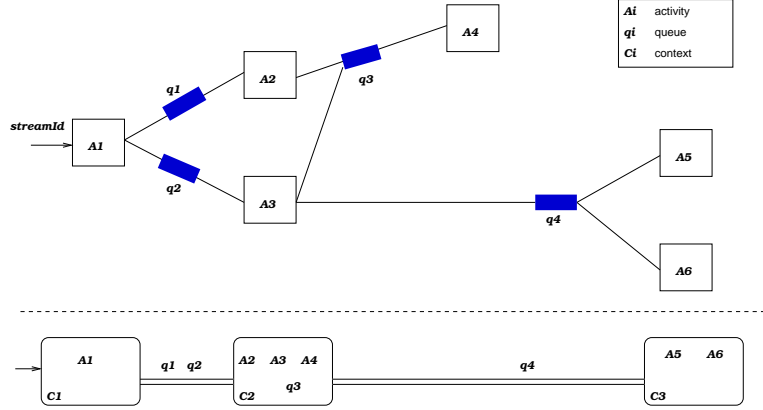


Figure 6: Control graph representing a service and its deployment on three processing contexts.

activities can be executed.

Typically, activation points are associated with data unit granularity, i.e. an activity can be executed only after the entire application-level data unit is delivered to the context. However, since incremental activities can operate on each data fragment separately, activation points can also exist at fragment boundaries. Specifically, if data is delivered to/from the context as a sequence of fragments with respect to which an activity is incremental, the activity can be executed ‘immediately after’/‘just before’ the data fragment is received/forwarded.

In order to illustrate this, we use the following examples. The activity that filters Delta events based on the flight number is incremental with respect to a 64B MAC-layer packet. This activity is invoked at the activation point that follows the receipt of each MAC-layer packet. For all packets preceding the one that contains the flight number, the activity will simply perform state updates. When the activity determines that it has the packet with the appropriate information, it compares the flight number with its parameters. If this fails, the activity will ‘consume’ the entire Delta event by discarding all MAC-layer packets that have been previously placed in memory, and all subsequent ones that belong to the same Delta event. An entirely different example is an application-level service, such as the Delta format translation, which is not incremental. This service’s activity can only be executed at activation points that exist after the entire application-level data is available in the context.

Service deployment. The deployment of a service includes (1) mapping the activities that represent the service onto activation points in the same or different contexts; (2) mapping the queues via which the activities communicate onto the inter- and intra-context communication channels that connect the corresponding application points; and (3) determining the stream identifier for the stream(s) to which the service will be applied.

Consider a service S , associated with a stream with tag t , and represented as a graph of activities (see Figure 6). For service S , activity A_c may be mapped to activation point A_p , associated with fragment f , if A_c is incremental with respect to the same fragment f . Next, communication channels must exist, that connect A_p to other activation points, where activities adjacent to A_c in S exist. The input and output queues of A_c are mapped to these channels. Finally, for each stream data unit, the value of the tag t must be accessible at A_p . The state maintained at A_p (look-up tables, rulesets, etc.) is adjusted to contain the specific values of t for which A_c executes.

To illustrate this we present the following example. A sample service used by Delta Airlines is one that performs format translation of business events pertaining to specific flight. The data stream processed by this service is a stream of Delta-internal flight events, identified through their format descriptor. This service contains an activity that performs filtering based on the flight number, which is incremental with respect to MAC-layer packets, and a format translation activity that operates on entire Delta business events. The deployment of the service requires that two activation points are identified, one with granularity of MAC-layer packets, and another that operates on entire application-level messages, along with the communication channel (e.g. shared memory pipe) that connects them. Once such activation points are selected, their internal state is updated to associate the Delta events' format descriptor with the corresponding activity (i.e., filtering or reformatting, depending upon the activation point). As a result, Delta internal events are filtered based on the value their 'flight number' field, as soon as they reach the first activation point. The remaining substream is reformatted at the second one.

Classification. Multiple activities may reside at an activation point. Each of these activities can potentially operate on more than one data tag. Therefore, at each activation

point, it is necessary to classify a data item or a data fragment, and determine which, if any, activity should be invoked. Since activities are part of a service that operates on a data stream, the classification mechanism requires the ability to (1) identify the stream identifier, (2) based on the classification rules, determine the corresponding activity, and (3) apply the same classification decision to the entire application-level data. The classification can be performed on data fragments or data units. The decision on how to classify the data is determined for each application-level data unit, based on specific fields in its tag. The same decision then needs to be associated with each fragment that corresponds to that data unit, without necessarily repeating the lookup process.

The subset of the fields in the data tag, based on which the classification is performed, is very application-dependent. The fact that there can be multiple activities at an activation point, associated with services that apply to different streams, whose identifiers are based on combination of communication- and application-level information, can make the design of the classification process very challenging [2]. For instance, for applications concerned with network-level services, the classification is performed based on protocol-related fields in the data tag, such as source and destination addresses, or protocol types. For the Delta application, however, the classification process is based on the format identifiers that define the business events exchanged in the system.

Furthermore, classification may need to be performed along multiple fields in the data tag. For instance, in order to apply the Delta filtering service only on Delta internal flight events for a set of specific destinations, the classification process needs to access both the data format and destination address fields in the data tag, and compare them against encodings for all possible format-address pairs. This can result in a large search space, necessary to maintain classification actions for all possible possible values of the classification parameters, and can make the classification process inefficient and time consuming. Therefore, our model specifies that the classification process associated with the receipt of data items (or data fragments) in a context, the default action taken by it, and the parts of the data tags which can be accessed by it, is determined by the application, based on the services that will ultimately be deployed at activation points in the specific context.

4.4 Concrete Implementation of the Model

The SPLITS programming system uses the model presented in Sections 4.1-4.3 to enable the dynamic deployment of application-level services on distributed systems composed of host-ANP nodes that have multiple contexts. Data stream sources and sinks can be external nodes communicating with the host-ANP node through the ANP-resident network interfaces, or application components executing in one of the host-ANP contexts. A data unit is defined by the application-level data exchanged between sources and sinks, and its tag consists of an application-level format description and network-level information identifying the source and sink. Each data unit is represented as a series of fragments that correspond to Ethernet RUDP frames. Furthermore, data is delivered to the receiving and from the transmitting ANP context as a series of MAC-layer packets. Fragments corresponding to these MAC-layer packets are also supported. The fragment tag for Ethernet messages corresponds to the first 16B in each frame that contain the Ethernet and RUDP headers. The fragment tag for each MAC-layer packet is the state value provided by the IXP1200 MAC interface that indicates the Ethernet frame to which the specific MAC-layer packet belongs.

An activity is represented as a stream handler, and a service is represented as a series of stream handlers executing in one or more host-ANP contexts. The granularities with which the current implementation of SPLITS associates activation points is on 64B MAC-layer packet, Ethernet frame, or entire application-level message (i.e., a data unit). The receiving and transmitting contexts have multiple activation points associated with fragments corresponding to a MAC-layer packet, an Ethernet RUDP frame, or an application-level data. All remaining contexts have only one activation point associated with application-level data units.

The incoming channels for the Rx context and the outgoing channels for the Tx contexts are the ANP's network interfaces, which deliver data at the granularity of MAC-layer packets. The communication channels between all other contexts on the host-ANP node are shared memory buffers. The queues connecting stream handlers in different contexts are mapped to these buffers. The communication channels between activation points in the same Rx or Tx contexts are represented by shared variables that track the transfer of

control between the protocol processing stages.

There can be multiple services simultaneously executed on an ANP. For each stream data, the classification process maps the data tag (stream identifier) to the service that needs to be invoked, i.e. to the stream handler that implements the service at that activation point. The deployment of a service involves either the deployment of a new stream handler at an activation point, or in the allocation of handler internal state for the data tags that define the stream. In both cases, a handler identifier identifies the mapping between a stream identifier and the stream handler. In this manner, the same stream handler can be reused to execute on behalf of different services.

Details regarding the implementation of stream handlers and SPLITS are presented in the following chapters.

4.5 Relationships with Other Models

The SPLITS programming model is sufficiently expressive to enable higher level programming models to be mapped to it. In addition, it represents a generalization of lower-level programming models, such as those operating at the network-level, and applications built with such models can be represented seamlessly.

4.5.1 Models for Network-level Services

The active networking model represented through the PLAN programming language [46], and Intel’s programming model [51, 54] developed specifically for the network processors used in our work are network-level models. In PLAN, active messages that correspond to network-layer packets represent data units, and they cannot be further fragmented. We deliberately support fragments such as lower-layer packets, since this is the granularity with which application-level data will become available in the processing context. Each active message is tagged with an active identifier, and a stream is a collection of active messages tagged with the same active ID. Activities are represented as active codes that are executed at activation points called execution engines. The classification process maps an active ID to the corresponding active code, and delivers the message to the appropriate execution engine. The active codes can only process messages tagged with one active ID.

Communication between execution contexts implies that messages are retagged with a new active ID, and reclassified. In our model, activities within the same context share context state, and avoid repeated reclassification. In order to implement application-level services with active codes, they would need to execute protocol processing in order to construct the application-level message, even if the activity is incremental with respect to an active message fragment. This is because the data tag used in PLAN, the active ID, maps data units to services, but does not bind active messages into an application-level message. If multiple service components are to perform different application-level processing, then each the application-level data unit would have to be reconstructed repeatedly.

Both programming models proposed by Intel, MicroACEs and microblocks [51, 54], support only network-level services, associated with the network-level headers of the packets that are handled. A network-level service is deployed as a collection of activities that execute on the separate processing contexts on the IXP, termed microengines. Data units correspond to network packets, and their tags to network headers. Activation points exist at the granularity of MAC-layer packets in receiving contexts, and at network packet layer granularity in all other contexts. A stream is a collection of all network packets to which the service can be applied. Packets tagged with headers for which the service is not supported (e.g., data from a specific source or from an unknown source, packets with different protocol types, etc.), are removed from the path, and are not processed by any of the IXP-resident activities. These packets may be directed to an external control context (such as the StrongArm/XScale core or the host) for exception handling, or simply discarded. All activities jointly execute one service, and all packets are processed by all activities. The model does not support distinct data paths through the contexts, nor does it permit the operation of a single activity to replicate the data unit and apply different processing to different replicas. The implementation of application-level services with this model is not feasible since classification is performed only on network packet boundaries and with network-level information.

The objectives of the PLAN and IXP models are to be expressive with respect to

network-level services, and to enable efficient implementations by optimizing resource allocation for the targeted domain. The implementation of network-level services will have better scalability properties compared to our model, since we provision resources for maintenance and manipulation of application-level state, processing, and content. However, the specific implementation of our model demonstrates that even with such increased expressiveness, we can meet the efficiency requirements for the application domain in which we advocate the use of host-ANP nodes.

4.5.2 Models Used with Streaming Applications

Higher-level programming models, such as those developed specifically for streaming applications [104, 27], or publish/subscribe models [30, 100], for instance, can be easily mapped to the SPLITS model. The stream identifier used in streaming models maps to the stream identifier used in our model. A data unit corresponds to each stream data item. Stream operators such as splitters and filters can be represented with activities that are associated with the appropriate data unit. Merging of streams can be represented either through use of the queue abstraction and an activity that performs the corresponding modifications. In addition, it may occur that the streams that need to be merged can be expressed with a single stream identifier, in which case the queue abstraction is not necessary. These models do not specify the possibility of associating stream operators with fragments of the stream data item.

Publish subscribe models like Echo [30] may be mapped as follows. A data channel is represented with an activity. An activity replicates each input data item for each of the subscribers. The stream identifier on the input stream represents a union of the tags of all data sources for that channel. For typed channels, the stream identifier also includes the data format identifier. The group of subscribers on a single channel defines its outputs. A new channel can be derived by creating a new activity, and ‘subscribing it’, connecting it to one of the outputs of the activity representing the original channel. The classification process maps the source of each data item to the activity that corresponds to the channel on which the data should be published.

Finally, our model can be used to represent frameworks for composable services, such as microprotocols [11]. The microprotocols are stackable components with well defined interfaces, and are used to represent complex services and their properties. Each microprotocol implements a specific property, and is defined as a collection of event handlers which are invoked when a particular event occurs. Using the SPLITS model, a microprotocol can be represented as an activity associated with a collection of data tags. Each data tag corresponds to the types of events that can be handled by the microprotocol. Different microprotocols are interconnected via event delivery channels, represented with queues. Finally, by enabling activities to operate with different granularity, the SPLITS model can be used to represent composite network protocols as well as composable application-level services.

CHAPTER 5

STREAM HANDLERS

Based on the programming model presented in Chapter 4, we define *stream handlers*, the basic computational unit of application-specific actions, to represent the activity applied to streaming data in the execution contexts belonging to the host-ANP nodes. In this chapter, we discuss the different types of stream handlers that can be embedded at various activation points along the stream data path through the host-ANP node, their interfaces and properties, and give examples of handler implementations. The goal is to present the possibility of using stream handlers to represent application-specific processing that can be executed in the fast path, and to also discuss the limitations of the approach.

5.1 Concept and Definitions

A stream handler is a lightweight, parameterizable, computational unit applied to stream data [40]. A stream handler represents an activity that can be deployed at an activation point on the data path through the host-ANP node, and executed in addition to basic actions like message assembly, fragmentation, and forwarding.

As with the activity it implements, a stream handler can be associated with data units or fragments of data units of specific types, i.e., with specific ‘tags’. A stream handler can operate on an entire application-level data unit, and (1) consume it, (2) modify it, or (3) generate new data item(s). In addition, stream handlers may also access and manipulate state that reflects the runtime conditions and the progress in the processing of the application-level data stream, at the concrete contexts where they are deployed. The exact actions taken by a handler at runtime are based on the data it processes, its type and content, and the internal state that represents rules, parameters, and progress information associated with the specific data type. Stream handlers may be parameterized, so that their operation is further determined by runtime checks on actual parameter values at activation

points.

Services are implemented with stream handlers that correspond to the activities of which the service is comprised. An activity can have multiple representations based on the activation points where it can be executed; therefore, multiple stream handlers can be associated with it, each corresponding to implementations at different activation points. Hence, a service can have multiple implementations that involve different stream handlers, depending on the activation points of the activities that compose it. These implementations may differ in their resource requirements and performance characteristics, due to differences in the properties and resource requirements of each of the handlers involved.

A rich set of application-level services can be implemented by deploying stream handlers to all parts of the data path, at any of its execution engines. On the ANP, stream handlers can be associated with the receive or transmit process of the data path through the ANP, or can be asynchronously applied to memory-resident application-level data.

Simple handlers implement network-centric functionality like routing and firewalling, executing code that uses network headers. More interesting stream handlers implement content-based filtering or mirroring, which means that they access payload data beyond the packet header and interpret the values of specific application-level fields, or they perform format transformations, which may require them to access and manipulate an entire message payload. Complex payload manipulations, however, are typically implemented by compositions of multiple stream handlers. For instance, the stream handlers implementing the *Access* functionality discussed in Chapter 2 (see Figure 2) range from simple ones that implement forwarding or filtering based on application-level content, to ones that perform data format translations. Handlers can then be composed to implement a service like format translation of stream data that meets some user-specific criteria, similar to the service evaluated in Section 8.3.5.

In addition, stream handler functionality can also be executed at the host, as kernel- or user-level handlers. This is important for complex services that require host-resident resources like floating point units or database accesses. At the host side, stream handlers

are represented by user-level functions and are invoked with the granularity of application-level messages, i.e. after an entire message has been received and assembled in the host's memory, or before it is passed to the send-side protocol code.

5.1.1 Accessing Application-level Data

Stream handlers require the ability to access, interpret and manipulate application-level information stored in memory buffers, or delivered as a sequence of network packets. To do so, it is necessary to provide handlers with information about the data layout and type or format, and to ensure that there is sufficient protocol support to guarantee that the result of handler execution impacts the entire application-level message.

Application-level data is described with compact binary data format descriptions [30, 17], which provide to handlers information about the structure and layout of the data they manipulate. The use of formats enables us to duplicate for packet bodies the elements that make it easy for NPs to perform header-based operations: known header formats, offsets, and types and sizes of fields. Stream handlers rely on such encodings to access the correct parts of the data or data fragment (e.g., MAC-layer packet(s), or memory buffers) on which they operate. Limitations of our work is that stream handlers currently use static encodings of data formats. Explicit data format representations (see [17] for a detailed description of these efficient binary formats) are used to obtain the correct field offsets, and to access the stream data as necessary, in a similar manner as other approaches use fixed header definitions to support header-based processing.

In order to execute application-level functions, stream handlers depend on the availability of underlying protocol code that assembles application-level data units, and the ability to interpret the structure of the data byte stream (i.e., protocol-related headers and packet properties) and access relevant packet body contents. For the current IXP platform, the IXP1200s, we rely on an RUDP-like protocol to efficiently implement the reassembly and fragmentation of application-level data in IXP memory on per flow basis [14, 60]. Our RUDP implementation is based on top of Ethernet frames, whose headers are extended to include fields such as message and frame sequence numbers, and End-Of-Message (EOM)

and Start-of-Message (SOM) bits to denote the first and the last Ethernet frame in an application-level message. We rely on RUDP because previous work has shown that TCP processing cannot be supported with the resources available on this platform [98]. For next generation IXPs, we expect to use standard protocols provided by the vendor.

5.2 Stream Handler Implementation

A stream handler represents an activity at a specific activation point. Therefore, its implementation concretizes the properties that define the corresponding activity. We next discuss how these properties have been translated in the stream handler implementation, the runtime state and operations necessary to enable the correct stream handler invocation, and the resource requirements they present.

Input streams. Stream handlers are associated with data units with specific tags, and the tags used in our implementation are the binary format descriptors discussed above. These format descriptors determine the data type of the application-level data units to which the stream handler can be applied, or the output types it can produce. The same format descriptor is also used to identify the data streams with which a specific service is associated, i.e. it also represents the stream identifier. For instance, in the Delta OIS both, the data type and the stream identifier is defined with the format descriptor of the Delta business events.

The SPLITS model also specifies that an activity can be associated with multiple data types, i.e. data items with different tags. A stream handler is represented at an activation point as a sequences of instructions, located at a specific address in the contexts' instruction store. Instances of the same stream handler applied to different input streams, i.e., data items with different tags, share their representation - sequences of instructions, but are distinguished by a different *stream handler identifier*. The stream handler identifier reflects the runtime mapping between the data tag, i.e. stream, and the stream handler - the activity executed on the stream data. That is, different stream handler identifier are associated with the same service when it is executed on a different stream.

Internal state. In addition, a subset of the handler's internal state is associated with each

stream. The size of this internal state is both application- and implementation- specific. It contains information maintained by each handler for each active stream, defined through the stream handler identifier, contains state regarding the processing progress, such as data counters, or partial results that are maintained across different data items. Part of the internal state are the parameters that further specify processing actions taken for that stream. For instance, a filtering handler will have a separate identifier and state for a stream consisting of FAA data vs. updates for external caterers, and maintain separate parameters for each one of them.

The implementation of the internal state is such that the handler can branch directly into the portion of state that is associated with the specific stream handler. This is maintained as part of the configuration state that translates handler identifiers into the location of the internal state with which the stream handler operates. In addition, internal state may be maintained for all instances of the same stream handler (that have different identifiers). This is necessary to associate handler parameters with all streams that can be processed by that stream handler.

Handler outputs. A stream handler is also defined by the type and number of outputs that it produces. In most cases, the output of a handler can be produced by directly modifying the original data item, and in such cases no additional memory needs to be allocated by the runtime. For instance, a filtering handler will simply forward the input data item, when the filtering action is not taken. The Delta format translation, and the layout of the Delta business event are such that the update event for caterers can be generated by directly writing over the input Delta data. The amount of memory required when the handler outputs cannot be produced from direct modifications of the input data, depends on the number and type of outputs. An example of such handler is evaluated with the OpenGL image cropping service, evaluated in Section 8.3.4. The output size of this handler is determined by the bounding box against which cropping is performed.

Queues. Stream handler that implements a specific service component, i.e. activity, is also defined by the abstract queues that connect its outputs to next activities in the service graph representation. These queues correspond to the communication channels between

the various activation points that are available in the runtime. The queues are defined by the data fragments that can be atomically enqueued and dequeued. A stream handler may not enqueue data on the queue, unless the entire fragment is available. For a queue that is defined for application-level data units, the entire sequence of fragments that represents the data unit has to be available, before it is enqueued.

A stream handler requires the ability to determine the specific communication channel to be used by its outputs. This is done by maintaining configuration parameters that represent the current mapping of the abstract queues to the data buffers used for inter-context communication. When a service is reconfigured so that a stream handler at a different activation point implements a required activity, the affected queue mappings need to be updated so as to use the appropriate supported communication channel.

External state. The external state required by a handler is represented through a set of state variables that track the processing progress on the data path, and configuration buffers that specify the service deployment. Both of them may be accessed by the stream handler code in order to determine the processing taken. This state is maintained as part of the context where the handler's activation point resides. Some parts of the external state may only be read by handler. For instance, the configuration state that represents the runtime service deployment are determined by the host application, and are used by the handler to determine the next context and the corresponding queue. Other parts, such as the progress in the data stream processing, may also be modified by the handler's operations. For instance, the stream handler may determine that a data item should be consumed, which is then reflected in the external state, so as to notify other stream handlers that may be involved in the processing of the same data.

Incremental activities. Finally, a stream handler which represents an incremental activity can operate on one data fragment at a time. These handlers can only be associated with activation points where data is delivered with the same fragment granularity. Typically, such activation points exist at the data receiving and transmitting contexts, where the fragment is defined by the underlying network layers. The fragments on which the handlers operate are associated with an application-level message and therefore, have the same

stream and handler identifiers. Hence, they are processed by the same stream handler, with the same state. The internal state can also be used to maintain partial results from the fragment processing.

5.3 Stream Handler Invocation

Next we discuss the implementation of activation points, the actions taken to enable stream handler invocations, and the state maintained.

On the ANP, the stream handler representation that can be executed in a particular context, resides at some specific location in the instruction store used by this context. The stream-specific state with respect to which the handlers actions are further defined, resides at a specific state location, typically represented as an offset within the handler's internal state.

Therefore, the actions taken at an activation points, necessary to invoke a handler on an input data item, are to determine the corresponding handler and state locations. In our implementation, the handler location is expressed as offset in the instruction store relative to a jump instruction. The execution of the jump instruction results in a branch to the first instruction in the stream handler code, and represents transfer of control to the handler. Upon completion, the handler returns control to the activation point (again through jump to a predetermined address). The outputs of its processing are then passed onto the appropriate queues.

5.3.1 Classification

Stream handlers are executed within activation points. The functionality executed at these activation points includes the appropriate runtime checks to determine the matching between the data tags and available stream handlers, to retrieve the corresponding parameters and additional configuration state, and to deliver the results from the handler processing to the appropriate consecutive context(s).

In order to invoke the appropriate handler, incoming data is classified based on the classification state (e.g., rulesets, lookup tables), maintained at the corresponding context. Figure 7 represents the basic functionality executed at activation points on Rx/Tx

or X threads on the IXP ANP. The call `get_sh_info()` performs a lookup in the context's configuration state to determine which handler should be invoked, if any. This state contains entries with values `[handlerId, handlerOffset, stateOffset]` that represent the stream handler instance invoked for the specific stream (stream handler identifier), its representation in the instruction store (handler offset) and its internal state (state offset). The classification process maps the stream data to an entry in this configuration state based on the stream identifier. Presently, we implement the configuration state in chip-resident memory. In addition, we support the use of hash tables to encode larger classification rulesets.

While the classification process used in our work uses only the data format descriptor to distinguish among streams, it can be extended to include additional information, such as data source or destination. This will pose additional requirements on the space size of the state used by the classifier and/or its implementation (e.g., multi-level hash tables).

5.3.2 Enabling Activation Points

The receive and transmit functionality on the host-ANP nodes is executed by the ANP-resident Rx and Tx contexts. The network interfaces on the ANP deliver Ethernet frames as a sequence of MAC-layer packets. The RUDP protocol code executed at the Rx contexts determines the sequence of MAC-layer packets that correspond to an Ethernet frame, and the sequence of Ethernet frames that correspond to an application-level data unit. Similarly, the Tx contexts execute the reverse operations. The data path through the host-ANP node may traverse all or some of the remaining available contexts, either on the host or the ANP, however, at all these context the entire application-level data is already available in memory. Therefore, at the Rx and Tx contexts we define activation points for activities that are incremental with respect to the MAC-layer packets and Ethernet frames. Stream handler implementations of activities associated with these two supported fragments can be deployed and invoked at the respective activation points. In addition, at all contexts, including the Rx and Tx, we define activation points where activities that operate on entire application-level data units can be executed.

Rx/Tx:

```
get_sh_info[apId, strmId, sh_id, sh_offset, sh_stateAddr]
.if (sh_id == NULL_HNDLR) br[continue#] . endif
jump[sh_offset, sh0#], targets[sh0#, sh1#, ... shn#]
sh0#:
    .....
br[continue#]

.....
shn#:
    .....

continue#:
    .if (msgState == EOM)
        check_new_config(apId)
    .endif
```

X:

```
dequeue(apid, msgAddr, queueId)
get_sh_info[apId, strmId, sh_id, sh_offset, sh_stateAddr]
jump[sh_offset, sh#0], targets[sh#0, sh#1, ...sh#n]

sh#0:
    .....
br[continue#]

.....
sh#n#:
    .....

continue#:
    .if (ctx == 0)
        .if (msgState == EOM)
            check_hot_swap(apId)
            check_new_config(apId)
        .endif
    .endif
```

Figure 7: Activation Points at Rx/Tx and X threads.

At each context, activation points require configuration state to determine the handler that needs to be invoked to process the stream data. In addition, the activation point provides the resources necessary for the stream handler execution. We next discuss the runtime support required to enable activation points at the respective points, the state maintained and actions taken which result in the appropriate handler execution.

5.3.3 Receive Contexts

There are three possible activation points which can be embedded within the protocol processing executed at the Rx context. The basic operations of the protocol code executed in this context, involve the following:

1. The first MAC-layer packet that corresponds to an application-level data unit is received.
2. MAC-layer packets are received until a complete Ethernet frame is received.
3. Ethernet frames are received until a complete application-level data item is received.
4. The application-level data unit is forwarded to the next context on the data path (e.g., host application waiting on receive, transmit context, etc.).

These steps do not show any header-based operations necessary to determine that the application-level data is correctly assembled from the MAC-layer packets and Ethernet frames. Namely, Ethernet frames and application-level data is assembled in memory by protocol code that operates on header fields containing sequence numbers, source and sink addresses, etc. References to the frame/message currently affected by the protocol processing are maintained in state variables `frmAddr` and `msgAddr`, respectively. Two other state variables, `frmState` and `msgState`, indicate whether the current operations involve the first and/or last fragment in the Ethernet frame or application-level data. In our implementation, the first and last MAC-layer packet in an Ethernet frame are identified through hardware support, and the first and last Ethernet frame in a message are distinguished through the values of the End-Of-Message (EOM) and Start-Of-Message (SOM) bits in the RUDP header.

The protocol code is extended with three activation points, associated with the receipt of a MAC-layer packet, an Ethernet frame, and an application-level data unit. The communication between handlers in these activation points is by passing a reference to the data or data fragment they require. The variables `frmAddr` and `msgAddr`, set by the underlying protocol, represent the corresponding communication channels. Synchronized access to these variables is enabled through `frmState` and `msgState`, respectively. For instance, data is delivered to the activation point associated with the receipt of an Ethernet frame, only when `frmState` indicates that the last MAC-layer packet which belongs to that frame has been processed. In order to determine the stream identifier, i.e. data tag, and invoke the appropriate handler, activation points also require the ability to determine the first fragment in an application-level data. A reference to the first fragment is provided through `msgAddr`.

There can be multiple messages ‘in transit’. State is required to map the protocol-level message identifier to the corresponding buffer (that resides at `msgAddr`) where fragments from the message are to be stored. Our RUDP implementation does not implement timeouts, and relies of the fact that all fragments will arrive before the message buffer needs to be reused. We can select one of the following choices: (1) if an SOM is detected, and the previous RUDP message is not fully assembled, it is discarded (the same happens on the start-of-Ethernet frame signal if the present Ethernet frame is incomplete), or (2) to maintain a window of number of messages that can be in transit, and then discard messages only when necessary, based on some policy (e.g., lowest sequence number). Both choices allow data to be lost potentially, but with different probabilities. In the first choice, the window size is one, and data delivered to the activation points is ordered. In the second choice, in order to guarantee that data is delivered to stream handlers in order, the state and parameters requirements may increase as $O(n)$ with respect to the window size. Alternatively, it may be specified that data can be delivered out-of-order to stream handlers, which will then have to determine internally the correct order based on network sequence numbers. This too can result in $O(n)$ state increase. Our implementation is based on the first choice. In addition, we deliver a sequence of fragments to the next stage whenever the last fragment in

the sequence is received, without verifying that indeed all internal fragments have arrived. This poses some additional constraints:

1. a handler implementing an incremental activity may need to reinitialize its state each time it starts processing a new application-level data; and
2. a handler may need to perform some form of ‘type checking’ to determine that the data received matches the format description (we maintain size for each sequence of fragments, but it may not be fixed since format can specify variable-length fields; therefore, ‘type checking’ has to be performed with application-specific knowledge available at the handler).

While these constraints are introduced due to limitations of the available protocol support, we believe that both should become the policy, rather than the exception, and that a handler should have the ability (1) to verify the correctness of the data - perhaps using some application-specific metrics, and determine ill-formed messages as those discussed in [43], (2) to discard the results of its current processing by consuming the incorrect data, or to implement some other set of operations when this condition occurs, i.e. introduce an exception arm in the service graph, and (3) to reset/reinitialize its state when necessary.

Examples of receive-side stream handling include content-based routing, filtering, or monitoring. Format translators of limited complexity may also be executed by receive stage threads. The goal of executing stream handlers with data fragment granularity is that based on the application-specific data formats, further optimizations can be performed on the data path, in the sense that a handler can be invoked as soon as the necessary data fields are received. In this manner, potentially unnecessary memory accesses can be avoided, per application-level data unit fast-path execution can be sped up, and higher throughput can be achieved.

5.3.4 Transmit Contexts

As the complexity of the stream handler and the amount of state information it requires increase, Rx-side processing is no longer feasible. An alternative to dedicating a separate thread to execute the stream handler, is to combine the execution of the stream handler

with the transmission process. Such transmit-side processing is useful whenever all or most of the data unit needs to be received before the handler can be executed. It is particularly efficient when data can be modified as it is being sent out, which reduces the loads imposed on ANP memory by avoiding double-copying.

There are three possible activation points in the Tx contexts: (1) when the entire application-level message is delivered to the context for transmission, (2) when the Ethernet fragment is determined, and (3) just before each MAC-layer fragment of which the Ethernet frame consist is passed to the network interface transmit buffers. Data is delivered to the Tx context through FIFO queues of addresses of memory buffers containing application-level data items. For each data item, the classification process at the first activation point determines the data tag, which is used at all three activation points to identify and execute the appropriate handler.

Stream handlers associated with the data transmission process, perform updates to memory/register locations representing data header or content, ‘just before’ it is sent to outgoing network ports. Multiple copies of the same data can be sent to multiple destinations, while maintaining a single copy of the application-level data in memory. This is useful for the efficient implementation of multicast customized based on destination. Again, explicit format representations are used to access the desired portions of the stream data unit.

On the IXP, we support two runtime implementation of the Tx context. First, one queue can be associated with each thread on the microengine executing the transmit side protocol. In this case, each thread presents a Tx context. The second implementation uses one queue shared among all 4 transmit threads, and the microengine to which these threads belong, represents a context. This enables more efficient implementation of data replication services. Consecutive data items are processed concurrently, in separate threads, where the stream handler executed by these threads implements the data replication. Thread synchronization guaranteed by the hardware round-robin, non-preemptive scheduling mechanisms.

At all activation points, a stream handler can also determine that the data should not be sent, i.e., the operation executed is to consume it. The decision to consume the

data can be driven by type checking or other application-specific decision implemented by a stream handler. Making a decision that a data item should be discarded in a stream handler execution in one activation point, may require the execution of some protocol-specific code to mask the missing protocol packets from the receive-side protocol code at the other endpoint. In our case, no action is taken upon message discard; the code simply skips over the remainder of that application-level message and retrieves the subsequent one. The start of message reset ensures that relevant parts of the internal state, such as packet and frame counters and addresses, are properly reinitialized. The receiving end will have to deal with potential partially received message. However, this can be detected and dealt with at the application level, using application-specific information.

A similar set of configuration state and variables to the ones used in the receive contexts is necessary to track and coordinate the transmission process with the different activation points. The Tx-handlers also perform state initialization when they start processing a new application-level data item, or the first fragment of which it consists. If necessary, type checking can also be performed. The destination address of the packets delivered to the network can be determined using handler-specific routing tables, maintained as part of the Tx-handlers internal state. However, it is possible that all Tx-handlers share the same 'global' routing table, or list of 'group' member addresses for implementation of functionality such as multicast, for instance.

5.3.5 Memory-resident Contexts

The remaining contexts on the ANP and on the host may also be included on the data path. Activation points at the X contexts on the ANP or at the host kernel- or user-level can run a wide variety of stream handlers, ranging from simple ones that implement forwarding or filtering based on application-level content, to complex handlers that perform data transcoding or format translation, and involve data copying. For applications that require complex stream handlers, it is necessary to dedicate separate X contexts to the execution of the handler's code. This is due to (1) the physical limitation of the instruction store associated with the microengines, and (2) the fact that a greater degree of concurrency

| | |
|--|--|
| <pre> Rx: .if ((cntPkt == 43) && (cntFrm==0)) .if (\$x0 == flight_no) discard_msg[msgAddr] .endif .endif Tx: move[fltno_offset, 43] dram_to_regs[\$\$x0, msgAddr, fltno_offset, 1] .if (\$\$x0 != flight_no) discard_msg[msgAddr] .endif </pre> | <pre> X: move[fltno_offset, 43] dram_to_regs[\$\$x0, msgAddr, fltno_offset, 1] .if (\$\$x0==flight_no) discard_msf[msgAddr] .endif </pre> |
|--|--|

Figure 8: Examples of Rx, X and Tx implementation of a stream handler.

on the critical path increases the sustainable rate of throughput.

Memory-resident transform handlers, X handlers, are executed on separate microengines that do not run any other protocol-specific fast path processing. These handlers operate on fully assembled application-level messages, and interact with the runtime through controlled access to the memory-resident FIFO queues available on the ANP. Each queue is identified by the context to which data is delivered. This information is used to map logical queues from the service graph to physical queues on the ANP. The mapping is determined at service deployment/reconfiguration time, by host-side application components, where information about both, the service and the processing contexts exists.. The configuration state at each context specifies the queues onto which the outputs from a stream handler processing should be placed.

Memory-resident handlers are less constrained in terms of the permitted operations and the order of accesses into the data payload, compared to the Rx and Tx handlers. Similarly to the Tx contexts, our implementation allows the X context to be configured as four X contexts concurrently executed by a microengine, each with a separate input queue, or as one X multithreaded context. Therefore, a stream handler implementation at an X context can use one or all of the microengine threads. Namely, each of the four microengine threads loaded on the specific microengine can process concurrently separate data units (i.e., application-level messages), or all four threads can jointly perform single application-specific task on one application-level message.

For instance, the data reformatting required in the Delta Airlines application for sharing Delta internal events with external catering systems can be implemented as four concurrent stream handlers reformatting Delta events. Thread synchronization and controlled access to queues is required to ensure correct execution.

More complex functionality, particularly when larger messages are concerned, can be sustained by allowing the handler programmer to use all four microengine threads to implement the data processing. This is the case with the image cropping handler used in an OpenGL pipeline, which performs simple computations, but accesses large portions of the application-level data. The particular example evaluated in Section 8.5 ‘touches’ all of the application-level message.

5.3.6 Resource Requirements

The properties of a stream handler establish its resource requirements in terms of memory and computational cycles. They can be divided into a set of compile-time and run-time requirements.

At compile time, resource requirements are defined by the size of the stream handler implementation at a specific activation point, i.e., the amount of instruction store required to represent the handler, the state that needs to be maintained for each of the data types that can be processed by the handler, its size, and the number, type and size of outputs that result from the handler processing.

The runtime requirements of a handler define its performance cost. The performance cost is a function of not just the compile time resource requirements at specific activation points, but also of a set of platform conditions (e.g., processing speeds, memory and interconnect speeds, etc.), and runtime conditions, such as current loads, incoming data rates, the sizes and frequencies of the different data types represented in the incoming stream, etc. Off-line profiling can be used to establish stream handlers’ performance cost under worst case, or other specific operating conditions, typical for the considered application [93]. The mechanism presented in Chapter 7 relies on off-line profiling to assess stream handlers’ cost, and to determine whether its deployment is permissible under the current conditions.

5.4 Programming Interface

Next we discuss how application programmers can develop stream handlers that need to implement specific activities. First, they need to define the application-level streams and the data tags of which the streams consist, in the context of the application's tagging system, e.g., data type, format identifier, network address, or some combination of these. Such global and per-stream information is maintained in well-defined shared memory locations. Next, they need to identify the different activities, the data tags they apply to, and the activation points at which these activities can be executed. Finally, they need to provide the stream handler implementation for each of the distinct activities, for each activation point, and identify the data tags for which the particular implementation is valid.

A handler is deployed at some specific offset in the context's instruction store. It is invoked with its `hdlrId` and has access to its internal state `stateOffset`. The stream handler's input is a stream data unit, or a fragment of it, each of some specific size. At invocation, the activation point provides the handler with a reference to the data it needs to operate on, and its state. The variables `msgAddr` and `msgState` deliver this information to stream handlers that operate on entire application level messages. It can access any portion of the application-level message by computing the correct offsets into the memory buffers pointed by `msgAddr`. Similarly, `frmAddr` and `frmState` provide the interface to stream handlers associated with Ethernet frame fragments. Rx-handlers that are invoked with MAC-layer packet granularity, use the variable `pktState` for state information. At invocation, the MAC-layer fragment is still partially in the receive buffers at the ANP's MAC interface, except for the first 16B, which have been read into the ANP's registers, and can be accessed through the transfer register `$$x0`. If the handler's operation results in a data item being consumed, it sets a variable `discrdDec` to notify the activation point of its output. Data replication is encoded through the use of `outQueues` and `bufferAddr`. Otherwise, the same variable `msgAddr` represents its sole output.

These variables are also used to keep track of certain external information, the status of the data flow on the path through the host-ANP node, and the message receipt and

Table 1: State variables available to stream handlers at different contexts on the IXP ANP.

| variable | description | Rx | Tx | X |
|--------------------------|-------------------------------------|----|----|---|
| <code>msgAddr</code> | address of current message | Y | Y | Y |
| <code>msgState</code> | tracks state of current message | Y | Y | N |
| <code>frmAddr</code> | address of current Ethernet frame | Y | Y | N |
| <code>frmState</code> | state of current Ethernet frame | Y | Y | N |
| <code>pktAddr</code> | address of current MAC-layer packet | Y | Y | N |
| <code>pktState</code> | state of current MAC-layer packet | Y | Y | N |
| <code>frmCnt</code> | counter for frames in message | Y | Y | Y |
| <code>pktCnt</code> | counter for packets in frame | Y | Y | N |
| <code>discardDec</code> | filtering decision | Y | Y | N |
| <code>outQueue</code> | next stage queue | Y | N | Y |
| <code>addrBuffer</code> | additional memory buffer | N | N | Y |
| <code>hdlrId</code> | stream handler identifier | Y | Y | Y |
| <code>stateOffset</code> | internal state location | Y | Y | Y |

transmission process, by maintaining counters for the number of MAC-layer packets received from a current Ethernet frame, number of Ethernet frames received from a current application-level message, and the address and status bits for the packet that is currently being accessed, as well as additional system state information. All of these are available to the stream handler programmer. Table 1 lists these variables, and indicates which contexts are they valid for.

In order to facilitate the development of stream handlers for the IXP, in addition to the Intel-provided macros, we provide a library of routines that is available to stream handler programmers for implementing the interactions with the host-ANP (SPLITS) runtime. These include macros for:

- moving data to/from memory, R/TFIFO and registers, such as `rfifo_to_regs()`, `rfifo_to_dram()`, `dram_to_regs()`, `regs_to_dram()`, `dram_to_tfifo()`, etc.;
- accessing handler state, such as: `read_hdlr_state()`, `write_hdlr_state()`, `copy_hdlr_state()`, etc.;
- accessing counters, `msgAddr`, `MsgState`, `pktState`, `pktAddr`, and other context specific state information maintained in global variables, such as: `read_var()`, `write_var()`,

`incr_var()`, `clear_var()`, `set_var()`, etc.;

- discarding the entire application-level message - `discard_msg()`, which affects the global variable `discardDec`;
- directing the output to specific next context(s) `set_queue()`, `enqueue_queue()`, etc.;
- and
- memory operations - `alloc_buffer()`, `free_buffer()`, etc.

Rx-and Tx- handlers can be invoked at the granularity of MAC-layer packets, Ethernet frames, or application-level messages. On the IXP1200, on the Rx-side, eight 4-byte registers are available for inspecting and manipulating data from the MAC-device' receive buffers. The firmware delivers only the first bytes of each Ethernet frame in registers, and the remainder of the frame is directly DMA'd to the correct memory location. A stream handler programmer can use routines such as `rfifo_to_regs()` and `rfifo_to_dram()` to alter the default behavior and access other portions of the frame and message. On the transmit side, the firmware invokes handler update routines at the beginning of each Ethernet frame it is about to transmit. A set of routines for accessing portions of the memory resident message are made available to the stream handler programmer, which are used to modify arbitrary portions of the message. Eight 4-byte registers are available for this process, therefore a single memory access can affect one DRAM line.

Unlike traditional application-level handlers, which operate solely on application data 'on top of' the underlying protocols, on the ANP, stream handlers are essentially combined with the protocol's execution. Therefore, the protocols used need to be taken into consideration, implying that stream handlers' programmers are aware of both data and header offsets, the fragment granularity that activation points are associated with, and of data delivery to/from the underlying network.

5.5 Summary

This chapter introduced stream handlers, lightweight, parameterizable computation units which can be used to implement a rich set of application-specific activities on ANPs. Stream handlers can be efficiently embedded at well-defined activation points on the data path

through the ANP, thereby enabling the joint execution of computations with communication. Due to the use of data formats to access, interpret and manipulate application-level information stored in memory buffers, or delivered as a sequence of network packets, stream handlers can be integrated with the receive- or transmit-side protocol processing. As a result, application-specific action can be executed with lower delays, unnecessary loads can be detected and prevented from affecting the system in a more timely manner, and even data increasing services can be supported efficiently by avoiding repeated copying and protocol stack traversals. Furthermore, by enabling services to execute on the ANP, applications benefit from the optimized NP hardware, its efficient support for large data movement and built-in hardware parallelism, even for computationally intensive processing performed on application-level data in ANP memory.

CHAPTER 6

SPLITS - SOFTWARE ARCHITECTURE FOR PROGRAMMABLE LIGHTWEIGHT STREAM HANDLERS

The following chapter presents the SPLITS software architecture which enables the use of stream handlers on the integrated host-ANP platforms and its components, and discusses the manner in which application-specific services can be dynamically deployed and configured onto the joint host-ANP contexts. SPLITS - Software architecture for LIghtweighT Stream handling, creates the activation points on the stream data path through the host-ANP nodes, where stream handlers implementing application-specific activities can be invoked. It also provides an interface to the application through which both the data path, and the processing applied along the data path can be modified dynamically.

6.1 SPLITS Framework

The SPLITS software architecture permits hosts to place selected communication services onto their ANPs, and to combine their execution with host-side application code. This enables a service to be (1) offloaded from a host node onto the ANP, (2) split across multiple execution contexts on an ANP, or (3) split across a host node and its ANP's processing engines. Offloading requires multiple service representations, suitable for host vs. ANP use, and splitting requires a service to be composed from multiple stream handlers that interact via well-defined interfaces, somewhat like the micro-protocols developed in previous research [11]. SPLITS permits handlers to interact across multiple execution contexts through controlled access to the shared memory buffers that contain application-level messages. We next describe the major components of SPLITS.

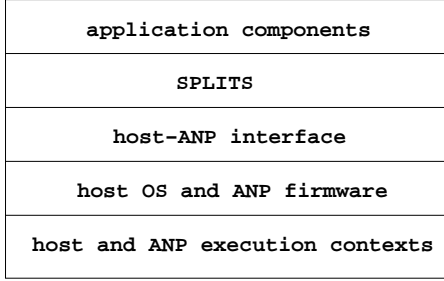


Figure 9: Position of SPLITS with respect to other layers.

6.2 System Components

In order to enable the use of stream handlers for execution of application-specific actions on the host-ANP nodes, and to utilize the programming model presented in Chapter 4 to develop application-specific services, we develop the software architecture SPLITS. SPLITS resides on top of the host and ANP firmware (see Figure 9), which, at minimum, establishes a shared communication channel between them. The objectives of SPLITS are to deliver the following capabilities:

- Identify the execution contexts on the host-ANP platform available to the application, and determine the default functionality that they implement. For instance, identify the contexts that are responsible for delivery of data to and from the host-ANP node (i.e., execution of the communication protocols), and the contexts that are available for execution of arbitrary application components.
- Establish data paths through the host-ANP nodes, which can traverse some, or all of the available execution contexts. This requires the creation of communication channels, such as shared memory buffers, between the distinct execution contexts on the host and the ANP.
- Identify distinct data streams, using application-specific classification rules, and associate them with specific data path through the host-ANP node.
- Distinguish the activities that need to be applied to a specific stream, determine the data fragment size for which each activity is incremental, and identify the contexts where it can be executed.

- Define the activation points residing in each execution context, and enable their runtime reconfiguration, in terms of the activities that can be executed at each activation point, the data streams that they are associated with, and their interaction with other activation points, in the same or in separate contexts.
- Support mechanisms to associate the appropriate implementation of each activity, i.e. stream handler, with the specific activation point, and to dynamically deploy or reconfigure it.
- Monitor the resource availability and performance levels at each execution context, for the data paths established through the host-ANP node.
- Use performance costs and resource requirements associated with different implementations of the same activity for specific stream and activation points, and dependent of the runtime resource utilization and the current performance requirements, select an admissible alternative.

Figure 10 depicts the host- and ANP-resident SPLITS components. The receive and transmit functionality is executed by designated contexts on the ANP. The Data Buffers represent the communication channels associated with each contexts, through which data can be delivered to it from other contexts on the host or the ANP. The Control Buffers represent the communication channels associated with the host and each of the contexts on the ANPs, and are used for exchange of control information related to runtime configuration or monitoring. The Data and Control Management components ensure synchronized and reliable access to the shared channels. The host side Application components can be executed by a single user-level context. In addition, not shown in the figure, there can be multiple concurrent execution contexts on the host, at both user and kernel levels. Information about runtime resource utilization is gathered and maintained by a Resource Monitor. The Constraint Verifier component drives the control mechanisms, and dictates the dynamic configuration of data paths and the processing applied along those paths, with consideration of the resource availability and requirements. The Control Manager implements the exchange of control messages with the ANP-resident contexts. The host-ANP interface, at

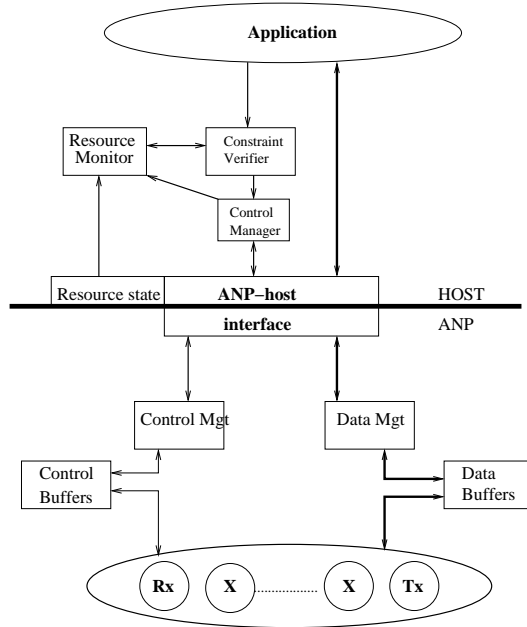


Figure 10: System components.

minimum enables the movement of data across the host-ANP boundaries, necessary for the exchange of both, stream data units, and control messages. In addition, it can export other runtime state information about the ANP or the host system which can be used for more efficient implementation of the control mechanisms.

In the following sections, we focus on the IXP1200-based implementation of SPLITS, and we describe in greater detail its components and the interactions necessary to enable the dynamic deployment and reconfiguration of application-specific functionality with stream handlers.

6.3 SPLITS on Host-IXP1200 Nodes

On the IXP1200 ANP, multiple microengine threads execute the receive- (Rx) and transmit- (Tx) side protocols, and/or implement the stream handler functionalities assigned to them. Our current implementation assigns one Rx and one Tx thread for each of the four Ethernet ports present in the IXP1200. Two microengines do not run any designated system components; their 8 threads can be dedicated solely to handler execution.

6.3.1 Data Buffers

All data packets manipulated on the ANP are stored in Data Buffers residing in ANP memory (we use SDRAM-resident buffers on the IXP1200). Our current implementation uses almost all of the available SDRAM memory for storing data packets. The buffers are fixed in size, and the address of each buffer, number of valid bytes, and additional parameters are maintained in buffer descriptors (or handles). The descriptors of all free buffers are maintained in a SRAM resident freelist. The Data Buffers are accessed via a set of SRAM resident queues shared by the ANP contexts, implemented as circular arrays of buffer descriptors. The SRAM queues are used both to coordinate the ANP's multiple execution contexts and to exchange data with host-resident handlers.

The SDRAM buffer size is fixed at compilation time. This implies that for messages smaller than the buffer size memory trashing will occur, and that larger messages will occupy several, potentially uncontiguous buffers. Since all data buffers constituting a single application-level message need to be passed to the next context, (i.e. enqueued/dequeued) atomically, buffer descriptors will need to be stored at each context until the full message is assembled, and the enqueue/dequeue overheads will be larger, due to longer locking period (more copies need to be made into the circular array), and additional checks necessary to determine whether all buffer descriptors belonging to the same application-level message have been enqueued/dequeued.

To minimize the negative impact of buffer-size mismatch, the buffer size should be set to a value that corresponds to the messages expected in the application. While this is not realistic in a general case, it is feasible in specific application contexts, such as those evaluated in this work. Another alternative is to maintain separate buffers for streams with different data size. Ultimately, a dynamic memory management utility should be developed, that will allocate and adjust buffer sizes to the specific messages.

6.3.2 Data Management

Data Management functionality includes the control necessary for (1) the shared access to the ANP-resident memory buffers, (2) their safe reuse, and (3) the exchange of data between

the ANP and the host, i.e. PIO or DMA transfers between the host and the ANP memory, also involving SDRAM data queues. We use two ANP threads to monitor and distribute locked access to the SRAM queues, and to ‘free’ unused DRAM buffers, and two threads to coordinate the host-ANP data movement, similarly to the host-IXP interface on top of which SPLITS is built [68].

Part of the Data Management functionality is the maintenance and controlled access to the SRAM-resident queues. The application-level messages stored in IXP memory, whether delivered on the incoming ANP ports or from the host, can be accessed by other execution contexts through these shared queues containing data buffer handles. The queue entries (1) can be dequeued by the Tx threads as messages are forwarded to their destination, (2) can be passed to the host node, or (3) can be processed by one of the memory-resident handlers on the IXP. The queues identify the path of a stream through the integrated node’s execution contexts, i.e., whether the stream is forwarded to one of the IXPs outgoing ports, directed to the attached host, or queued for additional processing at one of the deployed memory resident handlers running in separate threads. Each queue is associated with its sink context and is identified through a unique queue identifier. SPLITS has six default queues which include: four queues for the outgoing ports – TX0_QUEUE - TX3_QUEUE, a queue for the host – HOST_QUEUE, and a reserved queue, currently used for collecting runtime statistics – STAT_QUEUE. The implementation of SPLITS that uses one transmit queue for all four Tx threads has only one TX_QUEUE. Additional queues exist for each of the other available contexts, identified with Xi_QUEUE. These identifiers are mapped to bits in a 32-bit queue bitmap, which encodes the next context on the data path. The threads that monitor the queue accesses manage the locks for all updates to the queues’ head and tail pointers, which are maintained in on-chip memory. The queue identifier is used to index into this data structure.

6.3.3 Control Buffers

Interactions between the ANP and the application host node involve exchanges of control messages, similar to those described in [88]. For instance, a control message can contain

Table 2: Control message exchanged in SPLITS.

| | |
|---|--|
| [apId, streamId, hndlrId, qBitmap] | configures the data path by selecting an activity, and establishing the next contexts |
| [hndlrId, hndlrOffset, hndlrParams, hndlrState] | configures the activity, i.e. handler, by specifying its parameters or initialization state |
| [apId_new, apId_old, hndlrState] | specifies the contexts that need to be hotswapped, and the new object file representing its implementation |

parameters, one example being an identifier of the stream handler to be associated with a certain data stream. A small portion of the IXP memory is designated to implement SRAM- and on-chip-resident Control Buffers for enabling such control communications.

The Control Buffers are used for exchange of control messages between the host and the ANP. Table 2 provides descriptions of the control messages we support. These are divided into three sets, based on the tasks for which they are used. The first two sets are exchanged through SRAM control buffers between the host and each of the ANP contexts, and are used to modify the stream data path, the contexts it traverses, i.e., the next context queues onto which data is to be delivered, and the processing applied at activation points on this data path. The latter can be modified in two ways - a handler can be activated or deactivated, or new parameters can be passed to an already active handler. The last set of control message is exchanged between the host and the StrongArm core, and the operations triggered by these messages also require shared Control Buffers - mailboxes, between the StrongArm and each of the IXP's contexts for which hotswapping is enabled. The involvement of the StrongArm core is necessary since the host cannot directly load new codes onto the IXP. The dynamic hotswapping process is explained in Section 6.5.3.

6.3.4 Control Management

Control Management involves the exchange of configuration information between the host and some or all of the ANP's execution contexts. Examples include parameter updates

for ANP-resident stream handlers provided by the host and the transfer of monitoring information from ANP to host. The idea is to enable host-resident functions to direct the operation of stream handlers running in the ANP's fast path. Such control interchanges are implemented with Control Buffers manipulated via mailbox-based access functions.

Host-ANP interactions are implemented so as to avoid any unnecessary perturbations to the fast data paths maintained by the ANP. They utilize only a single, dedicated control thread on the IXP, which polls a mailbox shared with the host at some configurable frequency. Upon change, the control message is interpreted to determine the context to which it needs to be delivered, it is placed into the corresponding on-chip resident control mailbox, and the execution context affected by this change is signaled through a 2 bit semaphore-like mechanism. For instance, if a new Rx-side handler is to be enabled, then the control thread will signal the Rx-thread associated with the port to which the stream data is delivered. Similarly, a X context performing stream cropping will be signaled if new bounding box coordinates are to be used. The correctness of the concurrent accesses to the shared semaphore is guaranteed through the use of atomic bit-wise instructions, supported on the IXP hardware.

The configuration parameters are cached in IXP registers at each context, and are updated only upon change. The value of the semaphore is checked only at the beginning of new application-level messages. Memory access is required to retrieve new control values only upon host-initiated reconfiguration or upon switching among streams. The parameters values are copied in the context's configuration buffers for that handler, and the thread execution proceeds.

The control messages are set by the host, where information about the current ANP configuration and service deployment, as well as the application requirements are available. The IXP does not permit the host to manipulate the on-chip memory directly, therefore we use the indirect channel described above.

6.3.5 Host-side Components

On the host side, the deployment and configuration of handlers is managed by the Constraint Verifier. Depending on information about resource availability, provided by the Resource Monitor, the Constraint Verifier configures the data flow through the host-ANP node, as well as the multiple stream handlers which are to be applied along that path. Changes in application interests or in runtime operating conditions translate into requests to reconfigure stream handler deployments along the path; they are therefore, directed to Constraint Verifier. The host-side Control Manager is directly involved in the exchange of message with the ANP-resident contexts. These components are further discussed in Chapter 7.

6.3.6 Deploying Stream Handlers onto the Host-ANP Data Path

In the Rx and Tx threads, the stream handlers and activation point code, as well as the jump tables are fixed at compilation time, and are loaded jointly with the protocol-processing code in the instruction stores of the respective microengines. The memory-resident handlers can be compiled separately with code that implements the X-context's activation points, and can be deployed onto one of the IXP's free microengines at any time during the execution.

Fixed-size memory buffers are available to each handler for maintaining state across packet and message boundaries. Different handlers have different state size requirements, which can be met by designated registers, fast on-chip memory, or SRAM buffers. Currently, SPLITS stores handler state in statically allocated buffers, and we have evaluated the trade-offs of different state location options. Our future work will investigate the possibilities for dynamic allocation of different memories and state caching techniques.

6.4 SPLITS API

SPLITS enables us to dynamically deploy and configure stream handlers at activation points along the data path, thereby modifying both, the functionality implemented on this path, and the processing context it traverses. This section presents the interface available to applications to enable their interaction with SPLITS.

Our default configuration assumes that data delivered to any of the ANP's interfaces

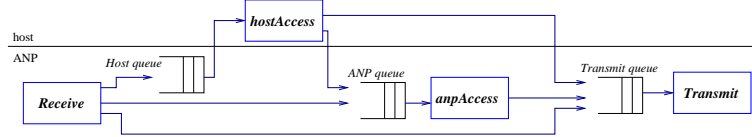


Figure 11: Data path.

(i.e., ports) is forwarded to the host-resident application. Similarly, all data from the host is forwarded to its destination through the ANP. The default host-ANP path is as follows. Rx threads on the ANP receive the data, execute the core RUDP-protocol functionality, assemble the application-level message, and enqueue it onto the host queue. At the host, the message is eventually delivered at the application layer, appropriate processing is performed, and, if necessary, the message is then further transmitted. The transmission processes packetizes the message into packets and enqueues it onto the appropriate transmit queues. The packets are ultimately sent out by the ANP's Tx threads.

Table 3 presents the SPLITS API used to reconfigure this data path. The API call `stream_define()` allows the application to identify the application-specific stream in which it is interested, by associating with it classification rules as pairs of tag fields and values. In this case, upon initialization, only data items from the defined streams will be delivered to the host-resident application component. Application-level data is passed to/from the application through `data_receive()` and `data_send()` calls that are built on top of the raw_sockets-based host-ANP interface, that implements of our version of the RUDP protocol.

Application-specific services associated with one or more of the defined streams are represented as sets of abstract activities. An activity can have multiple representations based on its execution context. The API call `service_define()` is used to establish the chain of activities that represent the service, and `handler_register()` is used to associate the corresponding activities with their stream handler representations for various activation points. This call determines the stream handler identifier, state and offset that is used by the SPLITS runtime.

The SPLITS API provides to applications the ability to reconfigure a stream data path by deploying stream handlers, i.e., to associate different handlers with different activation

Table 3: SPLITS host-side API.

| | |
|-------------------------------|---|
| <code>stream_define</code> | <code>(strmId, tagOffset, tagValue)</code> |
| <code>data_receive</code> | <code>(strmId, databuffer, size)</code> |
| <code>data_send</code> | <code>(strmId, databuffer, size)</code> |
| <code>service_define</code> | <code>(strmId, [actvId, apId])</code> |
| <code>handler_register</code> | <code>(strmId, actvId, apId, hndlrId, offset, initState, hndlrProfile)</code> |
| <code>datapath_config</code> | <code>(strmId, actvId, apId, [next_apId])</code> |
| <code>handler_config</code> | <code>(strmId, actvId, apId, params)</code> |

points on the data path. `datapath_config()` ensures that any new context is correctly included on the data path, and that stream data that needs to be processed by stream handlers at this context, is placed onto the corresponding queue.

The API also permits applications to change parameter values associated with activation points, thereby providing them with the ability to dynamically tune data path behavior. The idea is to enable direct application-level control of what operations are applied to the data path and how these operations are applied, thereby permitting the application- and client-conscious path behaviors critical to modern distributed applications. An activity can be reconfigured using the API call `handler_config()`, which delivers the new parameters to the execution context that currently executes it, or initiates the hotswapping process, if necessary.

In order to enable the functionality exported by the API, SPLITS maintains certain state about the application, and the current runtime operations. Table 4 represents each of these mappings.

6.5 Dynamic Reconfiguration

In order to best utilize platform resources and match current application needs, stream handlers need to be deployed and configured dynamically. While handler selection and parameterization rely on runtime resource monitoring, constraint checking, and resource

Table 4: State maintained by SPLITS runtime.

| Name | Mapping |
|------------|---|
| Streams | [strmId] - > array of currently deployed [actcvId, apId] |
| Activities | [actvId, strmId] - > array of possible [apId, hndlrId, offset, hndlrProfile] |
| Resources | [ctx, apId, actvId, headroom, istore, memory] |

management mechanisms, reconfiguration is triggered by the host node to which the NP is attached. It can be initiated in response to changes in system resources or in end user interests, and its primary purpose is to permit applications to adjust the behavior of the services they use. For instance, as client’s interests shift to different data in a remote scientific visualization, parameters can be used to identify the new coordinates of corresponding regions to be selected for forwarding to the client from each image in the stream. Or, as request loads increase on a server system, the host can change the level of data downsampling applied by the ANP-resident stream handler on a data stream, as permitted by the application.

The modes for dynamic reconfiguration supported by SPLITS are motivated by the physical limitations on the IXP architecture. Instruction store modifications are permitted only when the context associated with that instruction store is inactive, i.e. the microengine needs to be stopped before any instruction store changes can occur. This requirement implies that the services implemented by the stopped microengines’ threads would need to be interrupted, which can result in data losses, excessive buffering delays, or service degradation. For instance, existing approaches that enable low-level reconfigurability, such as the reconfigurable port extenders described in [103], during reconfiguration support only the baseline functionality of data forwarding. Similarly, on the IXP 1200s, with the Netbind [20] tool for dynamic network-centric datapath extensions (i.e., extensions that operate on the packets’ headers only), extensions are enabled by pausing the microengine while performing the reload, and the reported downtimes are on the order of several milliseconds.

SPLITS reconfigurability options give a much more flexible architecture, where reconfigurability and placement of application-level stream handlers is supported even in the basic datapath. In addition, it supports true dynamic code deployment and activation, with downtimes that are orders of magnitude smaller than those measured with Netbind, and are practically unnoticeable.

The remainder of this section discusses the reconfiguration options supported by SPLITS. Handler selection and parameter reconfiguration are the only two modes allowed on the baseline path, in the Rx and Tx threads. This introduces some overheads on the fast path, (evaluation results report minor), but gives reconfigurability options there as well. Dynamic hotswapping is only allowed in the X contexts.

6.5.1 Handler Selection

Applications can identify and dynamically choose among the currently loaded (i.e., the microcode is already compiled with the object file residing in a specific microengines' instruction store) handlers on the ANP.

On the IXP, the preloaded handlers are available to a single microengine only, since instruction stores are not shared. This is not a problem for the Rx and Tx threads, since the implementation of a single service that can be invoked at their activation points differ. However, if the remaining two microengines (or more (5) in the SPLITS design for the next generation IXP processors) need to share the pool of handlers, multiple copies of each handler need to be present in each instruction store.

A handler deployment can result in new context being included in the data path, or in changes at activation points at multiple contexts (e.g, moving the filtering functionality from the Rx to the Tx thread). Changes in the context traversed by the data path, i.e. other than the core Rx and Tx threads, are implemented by executing `datapath_config()`, which passes the appropriate control message to the corresponding context. The host-side Constraint Verifier ensures that the new target context has already been activated and the appropriate handler deployed, before making any modifications in the queue parameters.

A status field in the control messages used by SPLITS assists in synchronizing the modifications at multiple contexts on the data path. It tells the first context on the newly created/configured data path to mark the first message, so as to notify subsequent contexts that it is safe to retrieve the new configuration parameters. The marking is implemented by setting a bit in the first message's buffer handle. All subsequent contexts will not apply the new configuration parameters until the marked message is received.

6.5.2 Parameter Reconfiguration

The physical limitations of the size of the instruction store limit the number of stream handlers that can coexist as pre-loaded. In order to further increase the supported reconfiguration space, SPLITS permits parameters to be passed to currently active handlers, thereby tuning and customizing their operation. This allows the application to adapt the ANP's actions to runtime changes in application needs and operating conditions. For instance, in the SmartPointer application, based on the stages of the experiment and the scientists' interest in different types of molecules, the imaging server generates renderable images that represent different views of the ongoing simulation's output.

Parameters are passed from the host-side application component through the API call `handler_config()`, and the runtime ensures that the new parameters will become effective at the start of an application-level message. This avoids possible inconsistency due to processing of the same data with different parameters.

Our assumption, and a restriction on the programmer, is that the parameters layout in control memory is such that it can be interpreted in a handler-specific manner. Currently, a change in a single parameter will require that all other parameters are also re-written. Alternately, the parameters can be represented as an array of offsets and values, and additional interpretation can be performed to determine which exact parameter needs to be overwritten.

6.5.3 Dynamic Reloading

In order to best utilize resources and match current application needs and platform resources, stream handlers need to be deployed and configured dynamically (hot swapping of

handlers). While handler selection and parameterization rely on runtime resource monitoring, constraint checking, and resource management mechanisms, reconfiguration is triggered by the host node to which the NP is attached. It can be initiated in response to changes in system resources or in end user interests. Its primary purpose is to permit applications to adjust the behavior of the services they use. Simple reconfigurations involve changes to stream handler parameters, as with the remote graphics service described in Section 8.5 that adjusts image cropping to the current viewing area desired by an end user. More complex configurations require the dynamic exchange of the stream handlers used on a data path. Such hot-swapping must be sufficiently fast to avoid observable data path downtime, and it must be done such that state information (if any) is correctly passed from the old to the new handler.

Hot-swapping consumes additional ANP resources. In our current IXP1200 implementation, for instance, there are two microengines available solely for use by stream handlers. Our hot swap implementation uses one of these microengines to run current stream handlers while using the other one for hot swapping. This implies that the actual downtime for handler processing is equivalent to the costs of stopping one of the microengines and starting the other one. Measurements show that this can be done in about 28-30 microseconds. The newly loaded handler code loads any state information saved by the swapped handler from the memory. The drawback of this method is that one of the IXP1200's (few) microengines must be kept idle for hot-swapping. We expect this constraint to be less onerous in future IXP products.

6.6 Summary

The software architecture presented in this chapter, SPLITS, enables the dynamic creation, deployment, and configuration of application-specific services on host-ANP nodes. It defines activation points on the host-ANP contexts where application-specific activities, i.e. stream handlers, can be executed, and establishes communication channels which enable the data exchanges among these activities, necessary for the implementation of variety of services. SPLITS provides applications with the ability to configure services to be fully contained on

the ANP, or to span the host-ANP boundary. The latter is necessary in order to deal with the limited ANP resources, and sustain services which require computationally intensive tasks, or interactions with system components not available on the ANP, such as disks and floating point units, or software.

Finally, SPLITS addresses the dynamics in applications' behavior, interests, and operating conditions, by supporting mechanisms for service reconfigurability along several dimensions. Applications can dynamically select the contexts involved in the service execution, the stream handlers that implement one or more service components, or the specific parameters within which they operate. The specific implementation of SPLITS demonstrates that such reconfigurations can be executed efficiently and with practically negligible overheads, even when costly operations such as hot-swapping are required.

CHAPTER 7

SPLITS SUPPORT TOOLS

In order to enable the dynamic interaction of applications with the SPLITS runtime, SPLITS depends upon additional functionality implemented by a set of support tools. Such functionality is necessary to ensure that the newly deployed service can be executed within the platform resources, and that the runtime reconfigurations will not cause disruptions in the service-levels that are presently maintained. This chapter introduces these support tools, and conceptualizes the required information exchanges among the applications and the runtime, necessary for delivering the specified functionality.

7.1 Constraint Verifier

For reasons of programming complexity and system safety and security, end users cannot directly map stream handlers into ANPs. Instead, end users specify suitable stream handlers, but these handlers are ‘deployed’ only after being approved by the host-resident Constraint Verifier unit (see Figure 10). The Constraint Verifier is the main interface point for the application component that utilizes the SPLITS runtime, and all API calls are directed to it.

The main purpose of the Constraint Verifier is to determine whether a service reconfiguration can be performed, and to permit only those reconfigurations that satisfy a set of constraints. In order to do that, it requires the following functionality:

1. access information regarding the current system configuration - the application-level streams and the services registered with the runtime which manipulate these streams, the activities of which the services are composed and their set of alternate stream handler representations, and the processing contexts that are currently involved in their execution;
2. determine the service requirements and current operating conditions, such as incoming

- data rates, and current memory and computation loads;
3. determine the resources available in the runtime under these conditions, in terms of both memory, and excess of computational cycles that can be allocated at each context to additional stream handler processing, without degrading the service levels;
 4. perform constraint checking and assess the eligibility of a newly presented stream handler or handler parameters, to be deployed at a specified activation point along the data path, in terms of the service specification and the handler's resource requirements; and
 5. drive the execution of the control mechanisms and the exchange of control messages with the ANP resident contexts, which enable path reconfigurations in terms of both, the stream handlers executed on the path, and the contexts that the path traverses.

The application interacts with the SPLITS runtime via the API presented in Table 3. At initialization, the state maintained by the Constraint Verifier is initialized to represent the deployed SPLITS runtime, the defined data paths, and the preloaded stream handlers. All of the API calls used to reconfigure the runtime are directed to and executed under the control of the Constraint Verifier unit. In this manner, it is ensured that the appropriate updates can be performed to the initial state, and the Constraint Verifier will have an accurate representation of the runtime configuration, the data paths, the stream handlers applied along the path, and the activation points, i.e. contexts that execute additional stream processing functionality. The deployment of new handlers, and their reconfiguration via one of the methods presented in Section 6.5, is initiated by the application, but the Constraint Verifier unit prepares and triggers the exchange of control messages with the targeted execution context.

The primary objective of the Constraint Verifier is to ensure that any reconfiguration of the runtime is feasible and will not cause any disruption in the presently sustained service levels. The Constraint Verifier unit performs the necessary constraint checking to ensure that the reconfiguration requested by the application can be met within the resources available on the ANP. These constraints are verified by matching the runtime

resource availability and service requirements with the set of compile-time and run-time properties that define the handler. Unlike the host, we are particularly concerned with the ANP, due to its built in physical limitations. Therefore, the reconfiguration must satisfy several requirements. First, the stream handler's representation must fit within the instruction store limits on the ANP. Second, the memory required for the execution of the new stream handler, its parameters, state, and operational memory, must be within the memory designated on per handler basis by the SPLITS runtime. Finally, the run-time performance cost of the new handler must fit within the 'headroom' available on the specific execution context.

If ANP-level resources are not available, handlers are executed on the host, with a corresponding potential performance cost. Otherwise, the Constraint Verifier ensures the correctness of the reconfiguration, by generating control messages so that the appropriate contexts on the ANP load and/or activate the corresponding handler, and that any new context is correctly included on the data path. Furthermore, the actions implemented by these steps need to be performed atomically at an application-level message boundary, i.e. a single application-level message has to be processed either on the 'old' data path, or on the newly configured one. In addition, the Constraint Verifier ensures that each activation point that did not previously belong to the 'old' path is enabled with the appropriate configuration parameters prior to enabling the 'new' path.

Finally, the Constraint Verifier unit needs to enforce certain rules with respect to handler composition. For instance, one rule is that if the underlying protocol frame includes the data size, a Tx-side handler can be executed only if the new size is made available at the start of the application-level data, perhaps by a separate 'pre' handler, based on some runtime state. Other rules place restrictions on handler placement that ensure the correct implementation of the the application-level service graph. In general, rule sets are composed of multiple subsets. One subset is target-dependent, varying according to the specific ANP architecture used. Another subset concerns handler composition, to enforce certain inter-handler dependencies or data exchanges, for instance. A more general treatment of rules

and constraints applied to handler composition may be found in related work on micro-protocols [11].

The system state and available resources are maintained by the Resource Monitor. Therefore, each time a reconfiguration is permitted, the Constraint Verifier also notifies the Resource Monitor, so that the appropriate state updates can be performed.

7.2 Control Manager

The host-side Control Manager performs the synchronized mailbox accesses to communicate control message to ANP contexts (IXP microengines or the StrongArm). It delivers messages from the Constraint Verifier, and ensures that messages are delivered to all concerned contexts, in the appropriate order. For instance, it ensures that the first activation point affected by the path reconfiguration ‘mark’ the last application-level message before the reconfiguration actions are executed. Each subsequent context on the ‘old’ path will update its configuration parameters only after it processes the ‘marked’ message.

The Control Manager can also deliver resource updates from the ANP to the host-side Resource Monitor, however, the current SPLITS implementation does not support this type of data exchanges.

7.3 Resource Monitoring

In order to implement the designated functionality, and to allow the deployment and reconfiguration of stream handlers on the data path, the Constraint Verifier unit requires information regarding the resources available on the host-ANP node, under the current operating conditions and for the specific service requirements on each data path. The host-side Resource Monitoring unit (Figure 10) tracks the available ANP resources. For each ANP context, this unit maintains information about the currently active handlers, the available excess of cycles that can be dedicated to additional processing, and the memory that can be made available for handler use. Upon new handler deployment and/or parameter reconfiguration, these values are adjusted to reflect the current availability of host-ANP resources.

Our current model assumes static distribution of the available ANP memory that can be designated to active handlers at distinct activation points. In the multi-level memory hierarchy on the IXP NP, with each level the amount of memory which can be made available for handler processing increases with each hierarchy level, however, the accesses to it become more costly in terms of performance. At initialization time, application-developers should configure SPLITS to allocate the appropriate amounts and locations of handler memory, based on the application-specific requirements. For a specific SPLITS configuration, we also determine the total number of cycles in each context, that can be allocated for additional processing (i.e., headroom), which can be easily implemented with the existing IXP programming tools [50, 68]. These initial values represent the initial resource state on the ANP, and the Resource Monitor continues to update it.

Our current solution maintains information regarding the resource availability under most critical operating conditions, e.g., the highest data rates that can be delivered at the ANP network interfaces, and upper bound of the performance costs of currently deployed handlers. A more robust resource monitoring facility is needed to maintain a more accurate image regarding the runtime resource utilization on the ANP. Such a facility should deliver accurate and timely updates regarding actual data rates and processing and memory loads on the ANP, that are classified based on the individual streams that are currently handled on the ANP. The development of the mechanisms necessary to integrate such knowledge into the Constraint Verifier, and to ensure that we can respond to changes in these conditions in a safe and timely manner are beyond the scope of this work, and remain open for future research.

Our present implementation of the functionality supported by the Constraint Verifier is somewhat limited. During handler deployment, it performs constraint verification by comparing the required and available computation cycles at the target context. We do not implement a profiles database. Instead the required cyclecount is provided directly by the application, and is trusted to be correct. The Resource Monitor used by the Constraint Verifier tracks information about available headroom at each context, and the amount is updated whenever a new handler is deployed.

7.4 Stream Handler Profiling

In addition to assessing the available resources, the Constraint Verifier needs to compare these against the handler's compile-time resource requirements, i.e., memory and instruction store sizes, and to ensure that the handler's runtime requirements (performance cost) can be met with the available headroom on the stream data path. The available headroom is the excess of cycles available on the data path that can be utilized while still sustaining the same throughput levels. The handler performance cost is the amount of cycles it requires to process the data with which it is associated, on the current runtime, under the current runtime conditions. In order to avoid service degradation, this performance cost must fit within the headroom.

The task of determining the handler's performance cost accurately is highly complex, since it depends on variety of static and runtime factors. First, the computational cycles consumed by a stream handler depend upon a set of handler specific properties such as data unit sizes, fragment sizes it is associated with, number and types of instructions in the handler representation, etc. However, this number is further effected by runtime conditions on the ANP, such as data rates, existing computation and memory loads, etc. While the dependency upon data rates is more easily expressible, the affects of load fluctuations in different components, on a system with multiple parallel processing contexts, such as the IXP NPs, are difficult to model. Therefore, we rely on the use of handler profiles to associate an upper bound on the handler's performance cost.

Handlers can be profiled using off-line benchmarking, on hardware or using the vendor provided hardware simulators, such as the Intel simulator [52], and used to estimate upper bounds on handler 'cost'. Tools available for the next generation processors (ADT for IXP2xxx) offer even greater flexibility for off-line profiling, and can more easily used to evaluate the handler performance under specific constrains, such as message sizes, formats, percentage of data 'touched', etc. Such profiles are performed ahead-of-time for specific runtime conditions, and are maintained in a profiles database that is accessed by the Constraint Verifier.

Determining the performance costs of a stream handler under all possible runtime deployments and operating conditions is not feasible with such profiling mechanism. Such requirements will burden not only the profiling process, but also the interaction with the profiles database used.

Our current model uses only ‘worst-case’ profiles. ‘Worst case’, in this context, are profiles gathered under the most restrictive operating conditions and service requirements, i.e. when the ANP’s network interfaces deliver the maximum sustainable data rates, and when all execution contexts are processing the minimum-sized data items used by the application. Such profiles determine the upper bound on handler performance. While restrictive, this is sufficient to enable safe deployment of new handlers, while still satisfying the current service requirements, and meeting the platform’s resource availability. This approach can be extended by increasing the profiles spectrum with additional configurations that may be typical for a specific application, or by extrapolating the performance cost for handlers with parameters not represented in the profiles database, similarly to the use of profiles for task scheduling in cluster servers [93].

7.5 Safety, Security, Code Generation

The remainder of this section discusses issues that are not presently addressed in our work, but that can be addressed by future extensions of the SPLITS model.

Safety. Limited safety guarantees can be made through the use of compiler techniques and code verifiers to determine the validity of handler memory accesses and instruction store references. Solutions such as those developed for kernel-level extensions [81, 39, 8], that are sandboxing untrusted codes in order to isolate their behavior, are difficult to implement without additional hardware support for detecting protection boundaries. Implementing runtime checking with compiler-generated wrappers around all memory accesses will still not present a fool-proof solution, and will cause significant performance overheads. It is not clear whether strong safety guarantees can be made for arbitrary non-trusted codes for these platforms.

Security. In SPLITS, handlers are deployed through the host-resident Constraint Verifier

unit, and presently we assume that all applications interacting with this unit are trusted. The Constraint Verifier can be extended to include authentication mechanisms, which will enable it to identify the application, i.e. the principal with which it interacts. Access control rights can then be associated with different principals, and the Constraint Verifier will perform yet another constraint check, so as to allow only trusted applications to reconfigure the ANP runtime. Other access control policies can also be implemented in this manner. The Libra model also identifies the use of authentication and access control policies as a possible solution to addressing security related problems [100].

Code generation. The development of stream handlers is currently conducted by application programmers, which need to provide different handler representations for the various application-specific activities that may be deployed on the ANP. Compiler support can be used to automate this process. At each activation point, SPLITS defines the invocation interface for the stream handlers, and the granularity with which application-level data is delivered to the stream handler. Dynamic code generation for a single activity defined with some high level language for all application points is not possible, since certain activities may not have corresponding representations which operate with different granularity. Therefore, if in addition to the high level representation, the application programmer is allowed to specify the minimal granularity with which the activity can be performed, then compiler-level solutions can be developed that will generate stream handler code for the corresponding activation points. Finally, compiler-level support can be used to implement some of the safety mechanisms discussed above.

CHAPTER 8

EVALUATION

This chapter presents an experimental evaluation of this work using microbenchmarks in the context of several representative applications. It evaluates the overheads and the benefits of using hosts with attached network processors, the feasibility and the costs of implementing various application-level services with stream handlers on ANPs, and the performance and utility of the proposed software architecture.

8.1 Experimental Setup

The experiments presented are conducted using the aforementioned Radisys ENP2505 boards based on the IXP1200, interconnected via 100Mbps Ethernet links and via a Cisco 2980 switch. Each IXP board is attached to a host node in a cluster of eight Dell 530s with dual 1.7GHz Xeon processors running Linux 2.4.18. The host and the IXP NP are connected via a PCI interface, reported to perform at approximately 200Mbps in both directions [68]. We also use the IXP1200 simulation package SDK2.0, claimed to be cycle-accurate to within 2%.

The data streams used in the experiments are generated from several sources: (1) sequences of demo-replays of business data collected at Delta Airlines, (2) scientific data gathered from a molecular dynamics physics application used in the SmartPointer application [109], and (3) OpenGL data representing variable size images. The application components executed on the cluster nodes use our version of the RUDP protocol built on top of raw sockets. The same protocol is used on the IXP microengines.

We have evaluate the overheads of the use of host-ANP nodes and the viability of executing various application-level services on the IXP NP vs. on host nodes, by measuring the performance metrics along Paths a and b in Figure 12, respectively. In this configuration, Path b is directed directly to the host's Ethernet interface, and does not involve any of the

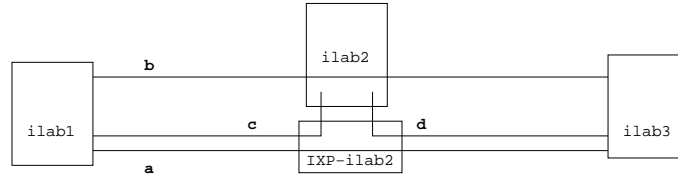


Figure 12: Different data path configurations used in experimental analyses.

IXP NP’s components. In addition, we evaluate the performance levels that can be achieved by implementing different services with stream handlers that execute jointly on hosts and on their attached IXP1200s. In these cases, performance is measured along Paths *c* and *d* independently, or jointly, along the Path *c-d*. These experiments evaluated the impact on load reduction on the host to which the IXP NP is attached, as well as the host’s ability to perform additional processing by exporting application-specific functionality to the ANP.

8.2 Improved Overlay Network Performance Using Host-ANP Nodes

The experiments presented in this section evaluate the overheads and benefits of using network processors attached to hosts for implementing the basic forwarding functionality required in distributed applications. They also justify our use of multiple activation points on the data path through the ANP.

8.2.1 ANP-forwarding Reduces Latency and Improves Throughput

As described in Chapter 1, typical configurations of distributed applications use overlay networks to implement their distribution functionality. The basic functionality of overlay nodes is to forward application-level messages, with the intent of using application-specific operations to modify data as it traverses the overlay. The results presented in Figure 13 evaluate the benefits of moving the basic overlay functionality from a host onto its attached NP. We compare the latency per application-level message for a host-side vs. an IXP-side deployment of a handler that implements message forwarding, both of which essentially entail performing a lookup for the next destination address. In Figure 13, both the host-side and the IXP-side forwarding are performed by a receive-side handler, which retrieves the next-hop address on the first packet belonging to a new application-level message. The

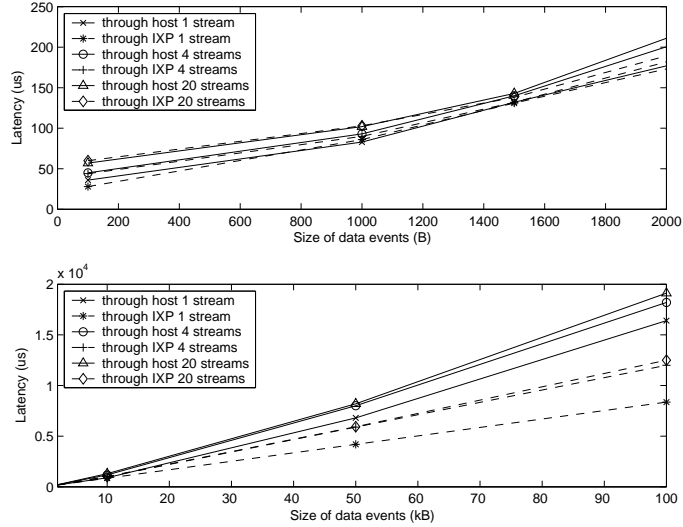


Figure 13: Data delivery delay for IXP- vs. host-level forwarding.

remainder of the message’s packets are then simply enqueued on the appropriate transmit queue. We perform multiple runs for streams consisting of 10,000 or 3,000 data items, for multiple data sizes (for large data sizes we used smaller number of data items per stream). Observe that for small messages, the IXP-level forwarding results in delays comparable with host-level forwarding (i.e., within 1%). However, as message sizes increase, the delivery delay for IXP forwarding is significantly smaller, with up to 30% measured decreases for tests with only a single stream. Latency reductions are due to efficient packet movements from and to the network on the IXP and the avoidance of costly kernel stack traversals on the host. These measurements also demonstrate that ANP-based forwarding scales better with respect to both increased data sizes and increased numbers of streams. The simple conclusion is that it is more efficient to implement the data forwarding performed in overlay networks on ANPs vs. on hosts.

In most application-level overlays, the forwarding decision is applied only after the entire application-level message is received. Figure 14 compares the throughput levels that can be sustained when such forwarding is implemented on the IXP, vs. on the host, where it is implemented on top of UDP. We report the end-to-end throughput attained on Paths a and b (see Figure 12). In addition, we include throughput measurements for direct UDP data transfers between two hosts only (Path b’), represented with the line marked ‘UDP direct’

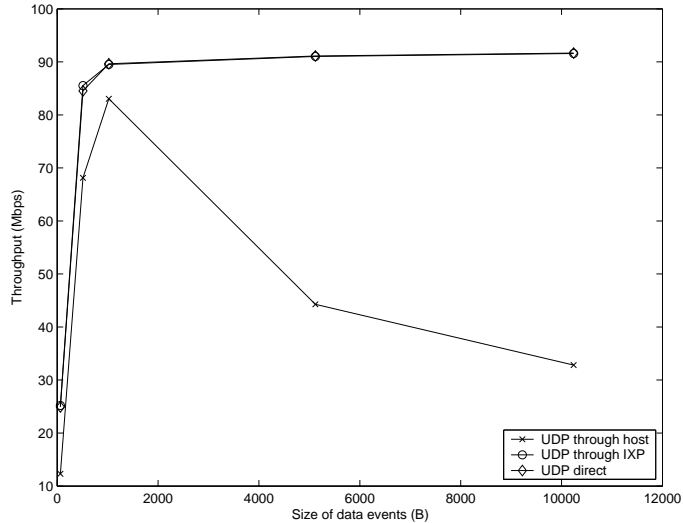


Figure 14: ANP vs. host throughput for UDP socket stream.

in Figure 14. We measure the throughput that can be sustained by forwarding 100,000 UDP packets of varying sizes through an intermediate host node, vs. an ANP. For the second case we modify the source’s ARP tables, so that data is delivered to the IXP’s interfaces. We notice that we can not only sustain higher throughput levels using ANP forwarding, particularly for larger message sizes, but furthermore, these throughput levels match the throughput that can be achieved with direct UDP communication between two host-side network interfaces.

The observed decrease in end-to-end latency and increase in sustainable throughput attained with ANP-side forwarding further motivate the use of host-ANP pairs in the overlays used by distributed streaming applications.

8.2.2 Importance of Multiple Activation Points on ANPs

The SPLITS model advocates the use of multiple activation points on the host-ANP data path, and more importantly, multiple activation points on the ANP alone, where Rx, X, and Tx stream handlers, operating on different data fragment sizes, can be invoked. Previous approaches, which implement additional functionality on an NP by operating on packet headers, exploit Rx-side processing by executing the handler’s code on each first MAC-layer packet of a new Ethernet frame [98, 20]. To demonstrate the importance of identifying

Table 5: Effect of handlers on throughput for different sizes of stream data items.

| data size \ handler | sh_1 (Tx) | sh_1 (Rx) | sh_2 (Tx) | sh_2 (Rx) |
|---------------------|-------------|-------------|-------------|-------------|
| 64B | 90.96 | 93.66 | 88.12 | 86.66 |
| 2048B | 99.45 | 99.98 | 98.82 | 84.14 |

multiple points on the data path through the ANP where a specific stream handler is invoked, we present the following results gathered with the Intel simulator.

The results in Table 5 demonstrate the feasibility of Rx-side processing in our implementation (Rx columns) for simple handlers ($sh1$), such as filtering stream handlers, which consist of few memory accesses and comparisons, and which require a limited amount of state to interpret the data format and maintain limited filtering parameters and state.

However, as the complexity of Rx-side handlers increases or as the data structures in the incoming data stream become more complex, and/or as the evaluation of filtering parameters requires multiple accesses in the packet stream and the maintenance of additional state, a point is reached at which Rx-side filtering is no longer feasible. Specifically, Rx-side processing has limited use when the transformations of the incoming stream depend on values of data fields embedded in subsequent MAC-packets, demonstrated by the throughput degradations observed when a complex headers ($sh2$) is used on the receive side (see Table 5, $sh2$ (Rx)). Rx-side filtering is appropriate only for simple handlers, which operate on small portions of the application data. In comparison, high throughput levels are attained with Tx-side filtering, even for complex filters that operate on an entire data message (Table 5, $sh2$ (Tx)).

The last set of experiments demonstrates the importance of transmit-side stream handlers, by evaluating an implementation of a per-client customized multicast. The graphs in Figure 15 represent the achieved throughput when the original stream is sent to n clients without any modifications (mcastO), when Rx-side handlers perform the n transformations needed to customize the single stream for each client and thereby, generate n customized copies of the original data (mcastRx), or when the Tx-side handlers simply customize the

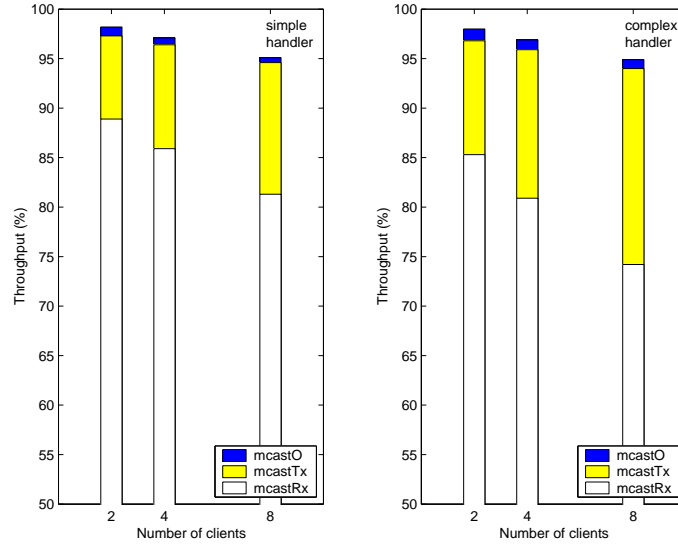


Figure 15: Client-customized multicast.

outgoing stream as it is being copied from memory and sent out (mcastTx). Note that such non-memory-intensive stream customization is possible only with Tx-side handler execution. Repeating this evaluation for two stream handlers that generate different levels of memory load, we find that the use of Tx-side stream handlers results in a per-client customizable multicast solution with efficiency within 10% of a non-customized multicast. The use of Rx-side handlers offering the same functionality, on the other hand, results in substantial degradation in the achieved throughput levels, offering performance that varies significantly based on the number of multicast clients.

These limitations motivate our design to support multiple activation points on the data path through the ANP, where stream handlers with different complexity and resource requirements can be executed. More detailed evaluations of the various tradeoffs between handler complexity and activation point can be found in [40].

8.3 Application-specific Services on ANPs: Feasibility and Limitations

The experiments described in this section demonstrate the feasibility of implementing application-level services of varying complexity using stream handlers on the IXP NP. They also evaluate and the costs associated with certain handlers, as function of their memory

and data access requirements.

8.3.1 Efficient Support for NP-based Services

This section compares the achieved throughput for host vs. IXP implementations of a set of services commonly required by distributed applications. The experiments are conducted by evaluating the execution of a set of application-level services implemented as stream handlers on the ANP on Path **a**, and a matching user-level representation of the same service on the host, along Path **b**. The results presented in Figure 16 show the measured throughput for the following services:

- data forwarding (route);
- mirroring of the original data stream to multiple destinations (mirror);
- filtering based on application-level content (filter);
- mirroring different views of the original data based on the destination (mcast);
- applying fixed processing for each application-level message, such as updates to set of data fields (delta); and
- applying complex customizations to the entire original stream data before it is forwarded (transform).

The IXP implementation of these services uses one or more stream handlers, executed by the Rx and Tx execution contexts only. This still leaves another two microengines unused (i.e., as available resources) on the IXP1200. There are no additional computational loads on the host CPU.

Throughput measurements are gathered for two modified data streams from an original stream gathered at Delta Airlines, so that they consist only of 1.5kB or 50kB data items. Observe that for both sizes of data items, the IXP implementations for data forwarding, mirroring and application-level filtering, deliver better throughput compared to their host-side equivalents. The cyclecounts for these services are much smaller than the available headroom in the context in which they are executing. This explains why sustainable bandwidth is not decreased.

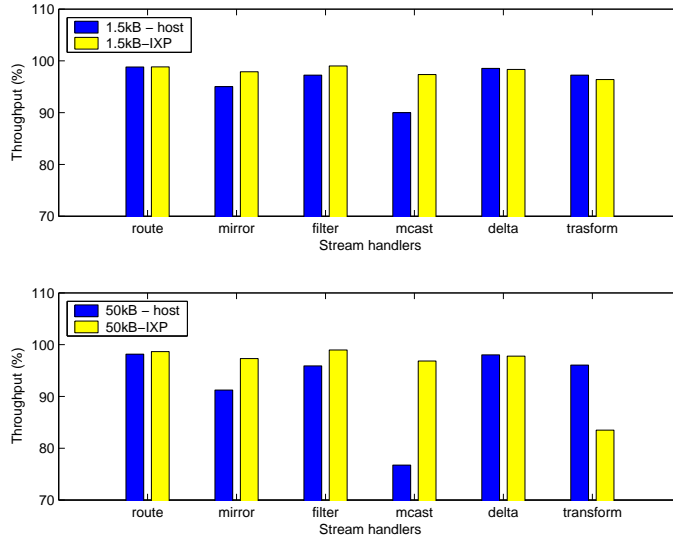


Figure 16: Handler complexity and sustainable throughput for set of services.

More interestingly, note the significantly higher throughput levels sustained for IXP-side implementations of data-increasing services, such as mirror or mcast. By executing these services as Tx-side handlers on the IXP, we avoid additional data copies, and we apply the data customization incrementally and as late as possible, just before its transmission.

Finally, these measurements show that while certain data transformations can be efficiently sustained on the direct Rx-Tx path (see the bars marked ‘delta’ in Figure 16), as the complexities of transformations increase, the throughput rates begin to drop, i.e., the cyclecount of the additional processing increases beyond the available headroom (see the bars marked ‘transform’). Furthermore, the ‘delta’ transformation has a slightly higher cost when applied to the smaller 1.5kB messages. This is a result of the smaller ‘headroom’ that is available on the fast path for smaller message sizes.

In summary, depending on the available resources, a variety of services can be efficiently supported on ANPs like the IXP1200. More importantly, data-increasing services proven costly when implemented on hosts, can be executed very efficiently when performed on the IXP. The performance improvements resulting from such ANP-side implementations are further enhanced by the fact that these services are off-loaded from the host, thereby permitting its resources to be used by other application components. We also show that the benefits of ANP-side processing are limited by the available resources and complexities of

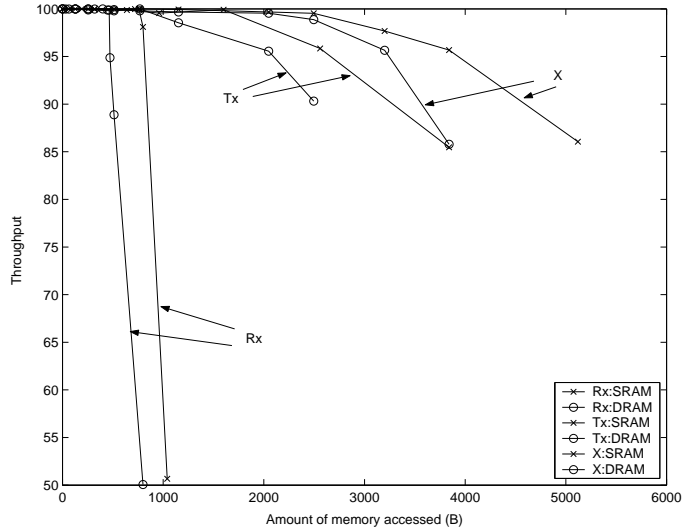


Figure 17: Stream handler performance cost as a function of the amount and type of memory accessed.

stream handler. This motivates our design of SPLITS and its support for deploying services across host-ANP boundaries.

8.3.2 Impact of Memory Accesses on Handler Placement

The stream handlers evaluated in the previous subsection represent basic functionality that can be extended to implement a variety of application-level services. One dimension in which the stream handler properties are affected based on the activity that it implements is the frequency and duration of memory accesses. Memory may be used to maintain handler parameters, or access and manipulate state or portions of the application-level data item. The results presented in Figure 17 represent the performance implications of deploying stream handlers with various memory requirements at different activation points on the ANP. In the experiments conducted, we vary the amounts of memory accessed by each handler, and the memory location - SRAM vs. DRAM.

Results indicate that there is a greater cycle budget associated with the X and Tx execution contexts on the IXP NP, and that stream handlers deployed at these contexts can execute larger numbers of memory accesses and, therefore manipulate more data, while still meeting throughput requirements. The amount of memory that can be accessed by Rx handlers is much more limited, and throughput levels drop sharply if this limit is exceeded.

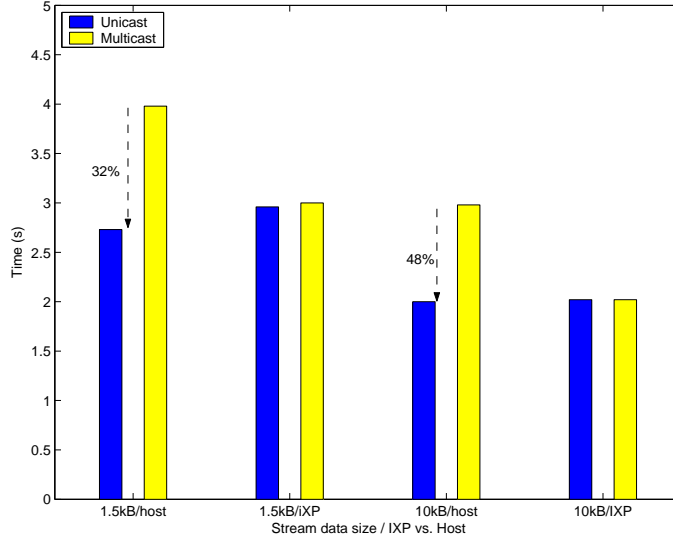


Figure 18: Performance gains for data increasing services.

Therefore, handlers with higher memory requirements should never be associated with the receive-side processing. In addition, results demonstrate the higher costs of DRAM vs. SRAM accesses, i.e. a smaller amount of DRAM memory can be accessed before the throughput levels begin declining. Consequently, both the amount and type of memory manipulations (e.g., state vs. data) should be taken under consideration when determining the appropriate activation point for a handler.

8.3.3 ANP-handlers Improve Performance of Data Increasing Services

The following experiment is conducted to evaluate the overheads incurred by data increasing services, such as replication or multicast. The measurements present the execution time to perform unicast, i.e., merely forward the application-level data stream, vs. multicast actions, that transmit each data item to multiple (in this case two) destinations. The results compare the execution times for unicast and multicast stream handlers deployed at the intermediate node on Paths a and b, at the IXP or host, respectively. The same tests are repeated for streams with data sizes of 1.5kB and 10kB. The total execution time represents the amount of time required at the destination to receive the entire data stream.

The unicast implementation is executed equally efficiently on the IXP as on the host, since they are both bound by the performance of the data source, whose data rate does not

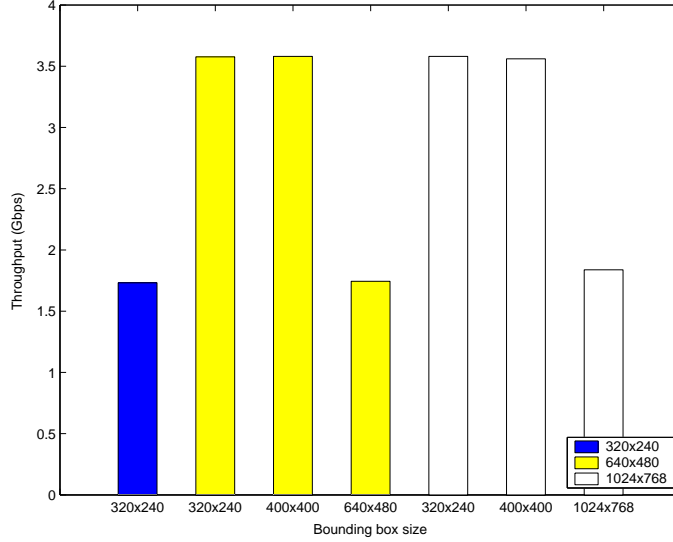


Figure 19: Evaluation of an image cropping handler.

fully stress the host’s or IXP’s capabilities. The multicast implementation on the IXP uses the hardware-supported contexts that execute the repeated data retransmissions concurrently, and without requiring additional data copies. Therefore, the measured performance for the IXP multicast handler is at the same level as the unicast handler. The host-side implementation of the same activity results in 32% and 48% increases in transmission times for the two data sizes evaluated.

The results show that implementing data increasing services such as multicast can deliver better performance to applications. Therefore, applications dependent upon such services, for instance, for achieving greater availability or fault tolerance, can attain significant performance improvements through the use of ANPs.

8.3.4 Memory-intensive Services on ANPs

The next experiment demonstrates the IXP’s ability to perform computationally intensive services, by evaluating a service used in remote graphics/visualization. This service crops OpenGL-produced images sent to clients using bounding boxes, and is computationally intensive due to the repeated memory accesses into the image data to perform comparisons for each 12B pixel. The idea is to experiment with the ANP’s ability to perform per-client data customization, with this service’s intent being to allow end users to reduce the

bandwidth requirements of incoming graphical data by selecting the viewing regions of current interest.

The experiments conducted here measure the bit-rates at which images of varying sizes can be cropped against certain bounding box coordinates, using the two free microengines on the IXP1200 running SPLITS. Parameters center the bounding boxes with respect to the images, which requires the stream handler implementing this service to inspect every 12B pixel to determine its relationship to the bounding box. The results presented in Figure 19 reflect the bit-rate at which the service itself can be performed, without including its assembly in IXP memory and defragmentation for transmission.

We evaluate the performance of the IXP-based cropping handler with data images preloaded in IXP memory, and we find that even lower-end platforms like the Intel's IXP1200 can efficiently perform such data intensive tasks at rates that reach 3.75Gbps (Figure 19). Similar behaviors can be observed for other stream handlers, such as ones implementing security-related censorship functionality, by not allowing a portion of an image's data to be viewed by individual end users, for instance.

The host-based implementation of the same handler, while performing satisfactorily with respect to sustainable throughput, requires 99.95% of the host CPU resources. This motivates service providers to move this type of functionality off host CPUs or server systems, thereby freeing their computational resources for other application components, while still maintaining service levels.

Two conclusions from these results are the following. First, even a complex service operating on most of the application-level message's payload can be implemented in the IXP1200 at bit-rates that far exceed the aggregate throughput this ANP can deliver. Second, it is important for ANP-level services to have access to application-level parameters, such as bounding box values. In this example, bounding box parameters significantly affect service times and throughput levels. More generally, parameter selection can be used to tune performance levels to current operating conditions.

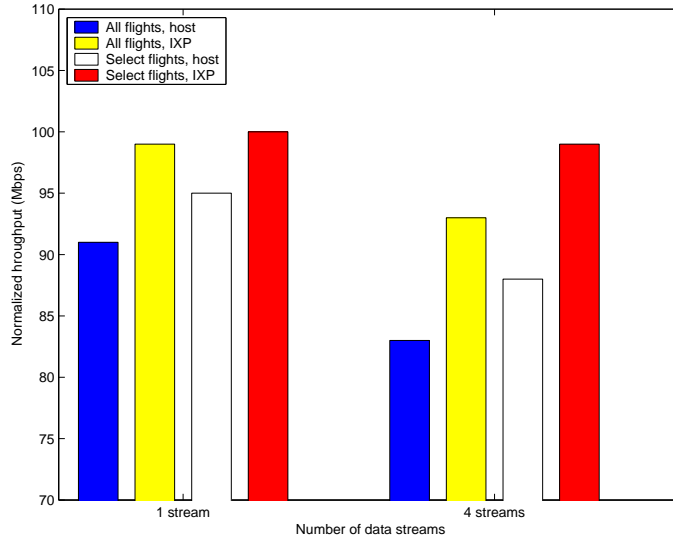


Figure 20: Efficient XML-based data transformation on the IXP. Benefits of parameter selection under increased loads.

8.3.5 XML-based Data Transcoding Stream Handlers Are Feasible on ANPs

Consider the operational information systems used in large companies, which interact with external service providers that cannot be given access to all of a company’s internal information. In the case of an airline, for example, when interacting with caterers, only information such as flight number, airport code, assigned gate and its anticipated change, and meal counts are exchanged. This information must be extracted from the internal flight events used in the airline’s information system, formatted and forwarded to caterers. The operational quality of such an application depends on the ability to efficiently perform such data format translations, on the timely delivery of data at predictable data rates, and the accuracy with which the newly formatted data matches client interests and capabilities. Therefore, it is necessary to selectively apply format translation functions, only to the necessary subset of the data stream. Furthermore, as services evolve, or as third-party providers change, efficient upgrades of format translation operations are needed, and such upgrades cannot disrupt or degrade the service quality offered by the system.

The results presented in Figure 20 demonstrate the ability of host-ANP pairs to efficiently perform certain data format transformations. We use a replay of binary representations of an original XML stream gathered from a large company. Its encoded data sizes

range from 360 to 420 bytes. The resulting events to be shared with the external client (e.g., a caterer) are 72 bytes each. Throughput measurements correspond to the rates at which data can be received and transformed, normalized by the sender's rate. The first two bars represent the throughput at which the host and the IXP, respectively, can perform the XML-based transcoding of all data in the original stream. The next two bars present the throughput at which only the flights originating at a specific airport are transcoded. This is accomplished by first applying a filtering handler, which discards all other events, and then applying the handler that performs the data format transformation. The same tests are repeated under increased network loads by adding additional Delta streams.

We observe, first, that in all cases, the IXP is better capable of performing services like these compared to the host. This is because it can efficiently access both header and payload data and extract only those fields required for the output format. The results in Figure 20 also demonstrate the importance of dynamic configuration (via parameter changes) under increased loads. The ability to extract from the original stream only the required events and to filter out the rest on the IXP, improves performance so that maximum data delivery rates can be met. The ability to filter out unnecessary data also improves the performance of the host-based implementation of the same service, but it still lags behind the IXP-based implementation by 12%.

8.4 Deploying Services on Host-ANP Nodes with SPLITS

So far, this chapter has demonstrated the feasibility of implementing a variety of application-specific services with stream handlers, presenting the performance costs and benefits associated with them. In this section we evaluate the use of the SPLITS implementation based on hosts that run standard Linux kernels and the Radisys IXP1200 ANP. We analyze the ability to dynamically deploy and configure new services on the data path through the host-ANP node, the impact of executing service components on the IXP on the host's performance, and the limitations of the current implementation.

8.4.1 Efficient Handler Deployment and Configuration

Experimental results indicate that the functionality in SPLITS which enables the dynamic deployment and configuration of services can be implemented efficiently. We do not foresee a requirement for high-rate reconfiguration requests, which means that experimentation focused on evaluating the end-to-end impact of the dynamic reconfiguration, and not the delay with which the reconfiguration request is executed.

Specifically, results indicate that reconfiguration overheads are practically negligible. That is, the overheads of performing the additional checks at activation points to support the dynamic reconfiguration do not impact the sustained throughput of the baseline functionality, and they add only negligible delays. The cost of actually performing the reloading of new configuration parameters depends on the preset parameters size (as discussed in Chapter 6), however, it is amortized over the subsequent stream data messages. Therefore, it does not impact the measured sustained throughput. The cost associated with dynamic hot-swapping is the downtime during which one microengine is stopped, the necessary state is copied, and the new microengine is started. Experimental measurements evaluate this time to be on the order of 28-30 microseconds, and has no noticeable impact on the end-to-end performance. Note that copying is required only for the relatively small amount of runtime state maintained in microengine registers, which is related to the processing progress in the data stream, such as message counters. The reconfiguration is performed at message boundaries, therefore, most of the registers can simply be reinitialized in the new context. All other state is maintained in on- and off-chip memory, and remains accessible from the ‘new’ microengine without additional copying.

These results demonstrate that we can efficiently enable dynamic reconfiguration of the application-level processing applied on the host-ANP nodes, with negligible overheads.

8.4.2 CPU Offloading

In some of the experiments conducted to evaluate our SPLITS implementation, we use standard benchmark applications to generate additional loads on the host node. Specifically, we used the `applu` benchmark from the SPEC CPU200 suite and the Linpack `n=1000`

benchmark [28]. This provides additional information about the increase in CPU availability as a result of both: (1) the offloading of services, or service components executed by stream handlers on the ANP, and (2) the reduction in the loads imposed on the host’s networking and memory subsystems when data filtering activities are performed on the ANP. The host’s CPU availability is estimated through the time, i.e. Mflops, that is available for the execution of these benchmarks. The measured results indicate two facts. First, CPU availability increases when stream handlers are offloaded on the ANP or when the amount of data delivered to the host is reduced by the ANP stream handler. Therefore, host resources are more readily available for execution of other application components. Second, the percentage increase in CPU availability exceeds the percentage decrease in amount of data. This is due to the removal of loads from the host’s networking and memory infrastructure. The result indicates that applications benefit from ANP-side service execution also due to the lower delay with which the service is invoked on the ANP, which in turn decreases the impact of specific events (e.g., unnecessary data, or ill-formed messages) on the remainder of the system.

8.5 Importance of Split Services Across Host-ANP Boundary

The second application with which we evaluate the proposed architecture is a distributed scientific collaboration. Data streams generated by a time-consuming, data-intensive molecular dynamics (MD) simulation are delivered to an imaging server. Based on the stages of the experiment and the scientists’ interest in different types of molecules, the imaging server generates renderable images which represent different views of the ongoing simulation’s output. The generation of these images is a computationally intensive task, involving floating point and matrix arithmetic. It imposes substantial loads on the server CPU and on its I/O and memory infrastructure. Once computed, images are forwarded to multiple clients or groups of clients. Depending on the clients’ networking and platform resources, or interests, the imaging server needs to further manipulate the image representation, such as downsampling the color encoding, performing cropping operations to match the client’s

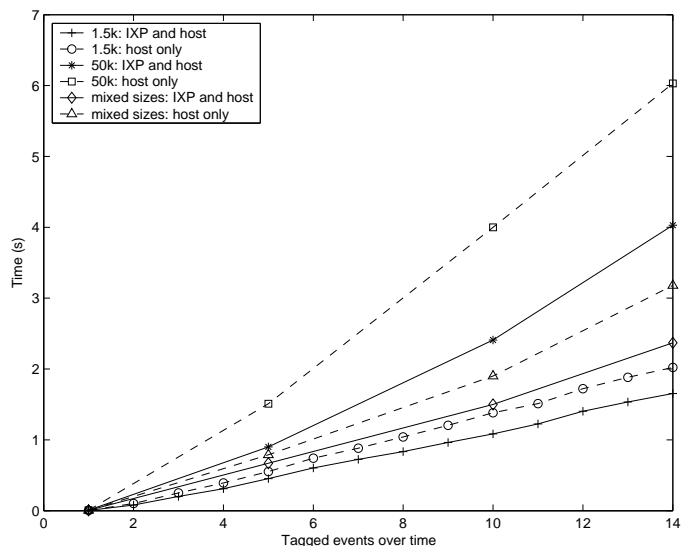


Figure 21: Importance of service deployment across Host-ANP boundaries.

view point, etc.

The experiments conducted using the SmartPointer application evaluated both, Path *c* - the delivery of molecular dynamics data to the image server, where its image representation is generated, and Path *d* - the customization of the generated image and its transmission to clients. The following subsections discuss our observations.

8.5.1 ANP Handlers Reduce Loads Delivered to Host Components

The graphs in Figure 21 demonstrate the performance gains attained by using ANPs to apply IXP-resident filtering stream handlers to select the subset of the simulation data stream of current interest to the scientists interacting with it. These measurements are gathered for three data streams: one includes the binary representation of the original molecular dynamic data with data sizes varying between 1.5 and 30 kilobytes, and the other two are altered MD streams that consist of 1.5kB and 50kB stream data. The service evaluated is applying the MD bond server computation to molecules of specific type only. For each data stream, we measure the processing times for a set of tagged events, in two cases. In the first case, the data stream is delivered directly on the host’s network interface, and both the filtering and the processing code are applied to it at the host’s user level. In the second case, the data stream passes through the host’s ANP, where the filtering handler

is applied to it, and only the resulting substream is delivered to the host.

These results demonstrate the importance of removing some service components, such as filtering, from the host, and onto the ANP: (1) it decreases processing times and (2) it reduces the latency perceived by the end-users for all of the three cases considered, with gains reaching 40% in the tests that involve 50kB data sizes (see top two lines in Figure 21). These gains are due to the offloading of the filtering handler from the host and delivering to the host only the data required by it, thereby eliminating unnecessary host-side protocol stack traversals and I/O loads.

8.5.2 Benefits of Offloading Even Non-communication Related Handlers

At the imaging server's egress side, we are concerned with graphics-related data manipulations. The results in Figure 22 evaluate the IXP's ability to perform computationally intensive image manipulation services on behalf of the host, by using a stream handler that crops OpenGL-produced images using bounding boxes. The stream handler used to implement the cropping functionality is the same as the one evaluated in Section 8.3.4.

The experiments conducted here measure the time required to crop and transmit the image. When this service is implemented on the network processor, the host sends the full image to the ANP, where cropping is performed, and the resulting image is sent out on the ANP's network interface. This same service is also implemented on the imaging server's CPU, and the performance of the two implementations is compared. In the experiments, we crop 1MB images against bounding boxes that reduce image size by 10 to 40 percent.

The results presented in Figure 22 reflect the performance gains due to the reduced execution time of the service when using the ANP for cropping. Gains are particularly visible when the host's CPU is heavily loaded (which is not unusual for the imaging server) and reach up to 40%. To simulate increased loads and assess the utilization of the CPU resources by the image cropping and sending, we run the standard benchmarking tool `linpack` [28] on the host, and record the Mflops that it reports to have available. The number decreases when the host executes the cropping, i.e. less resources are available for running other application-level codes. These results also indicate the importance of dynamically

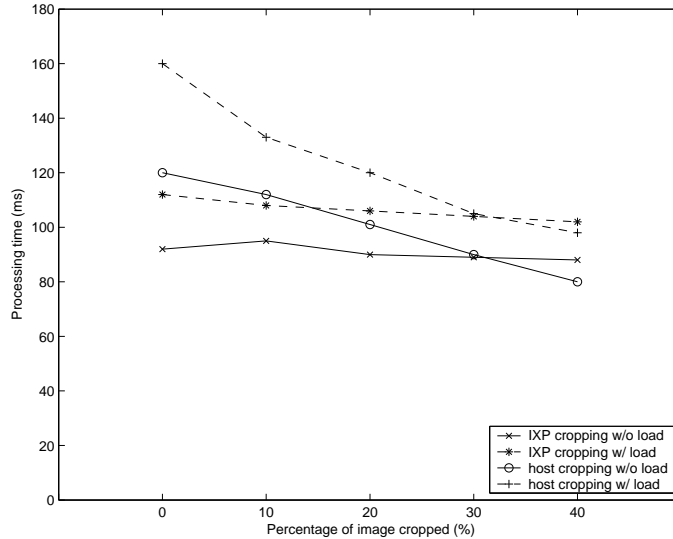


Figure 22: OpenGL pipeline service.

updating certain application parameters, such as the bounding box coordinates used in this experiment. This permits the ANP-resident service to better meet the end-user’s bandwidth and latency requirements.

Limitation of the PCI-based interconnect. Another observation from Figure 22 is the high penalty for using the PCI interface for host-ANP data movements. When cropping is implemented on the host, the CPU crops the image and sends a (possibly much) smaller image to the network device (via the PCI interface), thereby reducing total execution time. In the ANP implementation, the host has to send the entire image to its ANP, which limits the ANP’s cropping performance to the PCI throughput. Hence, as the cropping window size decreases, the performance of the host implementation starts increasing, whereas the performance of the ANP implementation is dictated by the data rates delivered from the PCI interface and does not change significantly.

With ANP’s becoming more powerful, it is highly unlikely that the ANP will ever be a bottleneck in the processing of application-level data. However, even if faster system-level interconnects replace the PCI bus used in our implementation, the movement of application-level processing onto ANPs needs to be done in consideration with the resource (i.e., bandwidth) availability at the host-ANP interconnect.

8.6 Summary of Results

The experimental results presented in this chapter demonstrate the utility, feasibility, and importance of implementing application-level services using NP-level stream handlers. Specific services used by a variety of large-scale distributed applications can be implemented with lower latencies and at higher throughput on ANPs rather than hosts. Services that perform poorly when executed on hosts at application level can be implemented efficiently, and ‘value-added’ services can be implemented at no additional costs as perceived by end users. Even complex services like graphics operations performed in OpenGL pipelines can be executed with high performance on an ANP.

Also efficiently executed on ANPs are data format translation and transcoding services, like those currently implemented with XML and XSLT in web services. Toward this end, we experimentally evaluate implementations of data format translations like those required in an e-commerce application.

Two other capabilities of host-ANP pairs are shown important. First, to be generally useful, it must be possible to have runtime host-ANP interactions, such as dynamic parameter selection to control the sustained performance levels. Second, our final results demonstrate the importance of ‘splitting’ services across host-ANP pairs and the limitations presented by the use of the PCI-based host-ANP interface. While already needed for tasks like runtime ANP control, the SPLITS architecture provides a model with which services can be mapped to host-ANP pairs, to effectively use their joint resources, and to dynamically cooperate to provide suitable end user functionality.

CHAPTER 9

RELATED WORK

The objective of SPLITS and stream handlers is to deliver to applications the dynamically reconfigurable communication and application-level services they require, so as to attain better resource utilization, to adapt more efficiently to the current operating conditions, or to address changes in application policies and end-user interests. Our approach is to create integrated platforms consisting of hosts and their attached network processors, and utilize the joint host-ANP resources. Services are represented as compositions of stream handlers which are created and deployed on hosts and on the ANPs, thereby extending the core networking functionality of these devices. The goals and the approach used in our work is related to many past and existing research efforts, in both industry and academia. With some we share similar objectives but differ in the context or the level at which solutions are developed. Others are related to us due to the use of similar target platforms, or the application domain addressed. This chapter surveys the related work, by classifying it into several groups based on the specific similarities with the work presented in this thesis.

9.1 Dynamic Service Customization in Streaming Applications

9.1.1 Middleware-level Customization

Extensive research in both industry and academia is targeting the delivery of ‘useful’ information to end users, by means of customizing the data source, or data stream itself [82, 3]. These approaches rely on peer-to-peer systems such as Chord [101], or Paste [29], or publish-subscribe infrastructures like ECho and JEcho, and Gryphon [30, 114, 113], to create the overlay networks used for the data distribution functionality.

The overlay used may be as simple as the front-end/back-end distinctions made in large-scale server systems [91, 36], or they may extend across multiple Inter- or Intra-net

nodes [6, 101]. Overlay networks [3, 101, 6, 22] have already established the utility of processing stream data ‘in transit’, for media transcoding [36, 90], for sensor data selection and fusion [35], and for handling the large data used in distributed scientific collaboration [109, 110]. Both peer-to-peer and publish-subscribe systems [101, 108, 113] permit applications to dynamically customize their data streams. For instance, multimedia data is routinely customized with data compression methods, and web applications manipulate data streams via web proxies, multicast methods, etc. [22, 72].

Our work is complementary to this research. Our goal is to offer such systems performance benefits by mapping some of the middleware- and application-level services they require onto programmable network processors. This is particularly important, since application-level implementation of data stream customizations suffers from significant overheads due to the repeated protocol-stack traversals to and from the application layer, and the loads imposed on the host’s CPU and I/O infrastructure. Furthermore, experimental results presented in this thesis demonstrate that many application-specific customizations can be executed more efficiently on ANPs.

9.1.2 Kernel-level Extensions

The performance limitations of delivering dynamically customizable application-level services in high data rate applications, has motivated the development of kernel-level support for specialized services [81, 39, 78, 8]. These approaches typically rely on the availability of safe, yet expressive subsets of standard languages for implementation of application-level services, and the underlying OS support for dynamic linking and safety checking at the host. Our future work will consider the use of language support to provide higher level of safety in SPLITS as well.

Compositions of fine-grain modules, along with dynamic code generation and software feedback, is used in the Synthetix project to achieve specialized kernel-level optimizations, e.g., in the communications protocols, which will result in optimal performance under given system conditions [81]. Such specializations can be used to adapt the functionality implemented at different nodes in the system to the current network conditions, and to achieve an

efficient scheduler for placing data on the network, for instance. We believe that exchange of feedback information can be used to tune handler parameters.

Similar solutions to efficient and safe kernel-level specialized services are developed in the other efforts such as SPIN or KPlugins [39, 8], or the KECho kernel-level event delivery system, needed to support high data-rate applications built on top of publish/subscribe infrastructures. Kernel-level application-specific service extensions can be used at host-ANP nodes, to create vertical platforms that offer to applications more tradeoffs in performance vs. resource availability, required to address a larger set of service requirements.

9.2 Modular Frameworks and Service Compositions

Stream handler compositions are similar to those performed in modular frameworks for dynamically generating higher-level services from simpler components [67, 11, 80, 96]. Unique to our approach is the deployment of these components (i.e., of stream handlers) across the host/ANP boundary, depending on current service requirements and available system and network resources.

Existing research has already evaluated the use of composing complex and/or customized computations and/or properties through modules. Coyote and Ensamble are examples of modular frameworks for building configurable, higher-level services from microprotocols [11, 67]. For instance, in [47], Hiltunen et al. use microprotocols for constructing highly configurable fault-tolerant distributed services. The motivation for offering composable services in these systems is to optimize the data path so that it includes only the modules required by the application. Cactus is an instance of the Coyote framework which supports customizable, dynamic quality-of-service control [48]. Our research would benefit from such work to construct modular and perhaps, dynamically composable higher-level protocols that could more flexibly leverage application semantics when dynamically adapting event mirroring and the levels of consistency and synchrony maintained across replicated commercial servers.

CANS [38] and Active Streams [18] are two examples of application-level infrastructures that support dynamic service compositions by injecting application-specific components into

the data path through the network, and adapting them to the system conditions. These and other similar approaches are concerned with the ability to share and reuse service components, and deliver application-specific services, capable of run-time adaptations, necessary to deliver end-to-end quality-of-service. The experimental results in Chapter 8 demonstrate that NPs can be used to execute many application-specific service with better QoS properties than standard host. Our work is similar to these approaches in that we also consider the deployment of service components to deliver more efficient services, however the paths that are primarily addressed by SPLITS are those internal to the host-ANP node. In addition, developers of SPLITS applications can benefit from stream handler repositories, similar to the component repositories used with these, and other component-based approaches.

Protocol Boosters [33, 96] is another modular framework for flexible creation of communication protocols from components. The framework relies on programmable network infrastructure to insert the protocol modules along the data path through the network, and realize the customized communication. The host-ANP nodes proposed in this thesis can deliver this infrastructure to applications. Other modular frameworks targeting network services, include the Click modular router and the Scout OS [56, 69, 76], which target the creation of high-performance networking paths using general-purpose processors, or the Dynamic Hardware Plugins, developed by Turner et al. [103], which utilize hardware plugins (Port Extenders) implemented as reconfigurable logic, to extend the core functionality of a Gigabit switch.

Several research projects concerned with composable frameworks include extensive support for verifying the correctness of the composite service or protocol [67]. As part of the Ensemble project, formal specifications for the micro-protocols have been developed, which are used for verifying correctness and perform optimizations of the protocol code. SPLITS can benefit from similar formal specifications for stream handlers, in order to improve the safety guarantees which can be delivered to applications. The availability of such tools is particularly critical for the ANP, due to lack of hardware supported protection mechanisms.

9.2.1 Split Services

Complementary work demonstrates the utility of executing compositions of various protocols vs. application-level actions in different processing contexts. Examples include splitting the TCP/IP protocol stack across general purpose processors and dedicated network devices [15, 85, 87], or splitting the application stack as with content-based load balancing for an http server [7] or for efficient implementations of media services [90]. Our work addresses the benefits of split service implementation across multiple execution engines. Furthermore, we generalize specific solutions to arbitrary sets of services by designing a software architecture and support for splitting services across hosts and their attached programmable network processors. The intent is to benefit from the efficient data movement mechanisms available on network processors, from their ability to access packet header and payload data at low costs, and from their built-in parallelism.

9.3 Extensible network infrastructures

9.3.1 Device-level Research

There are numerous interface designs for closely coupled network devices, such as programmable network processors or line cards, to host nodes, using OS-controlled mappings [88, 102, 103, 96]. The motivation is to take advantage of the ‘network-near’ nature of these devices vs. hosts, and implement transaction services, synchronization functions and service- (e.g., for RPC) or system- (e.g., for Linux) specific optimizations of protocol stacks and of the data movement and buffering associated with message communication. Similar argument is made for use of specialized devices for attaining intelligent disks, that can execute application-specific I/O functionality with much improved performance levels over standard hosts [55, 59, 1]. SPLITS has similar objectives, since it aims to deliver performance benefits to applications from the use of specialized hardware on the ANP, to offload hosts, and to enable more efficient implementation of some services by coupling computations with communication.

9.3.2 Active Networks

Research with active networks has explored the possibilities of extending and customizing the behavior of the network infrastructure to meet application needs [106, 62, 5, 92, 102]. The use of stream handlers on attached network processors is similar to active networking approaches in the sense that it aims to exploit the programmability in the networking infrastructure, and to dynamically associate application-specific codes with select data flows.

The use of active identifiers in the active messages and capsules, in systems such as ANTS, CANEs, and Libra [100, 92, 106], for determining the activity which needs to be applied to specific packets is similar to the mappings between the data tags and the stream handler identifiers used in SPLITS. The Execution Environments and Active Applications used in the Libra framework for developing composable active network services [100] is similar to the activation points and the stream handlers invoked deployed at these points in our work.

These approaches provide an extremely general mechanism to adapting communications, and raise many safety and security concerns. In order to address these concerns, while still meeting the data rate requirements at the targeted shared networking devices, the active networking approach imposes restrictions on the service complexity and on the data accesses it can perform. Therefore, it is better suited for network-level services, performing network header-based operations.

We differ from active networking approaches by focusing on the ‘edges’ of the network, and by forming closed, controlled environments consisting of the host and the attached NP. In this manner we avoid many of the safety and security issues raised in the context of active networks, and still offer the needed dynamically customizable services to applications.

9.3.3 Use of Network Processors for Application-specific Services

Our work is related to existing research that exploits the programmability of network processors in several different domains. Earlier work with programmable network processors mainly focuses on providing network-centric NP functions based on the header contents of the incoming traffic. Examples include NP-level software routing, firewalling, low-level

intrusion detection, packet classification and scheduling, network monitoring, software multicast, and service differentiation across different traffic classes [98, 20, 116, 56, 63, 102]. Application-level services previously realized on NPs include load-balancing for cluster servers, the proxy forwarding functionality servers need, and payload caching [7, 112]. These approaches are focused on delivering performance improvements to web servers, and the implementation the NP-level functionality relies on access to the http header. With our approach, these and many other services can be implemented on ANPs or jointly by ANPs and hosts, independent of the specific protocol- or application-level headers being used.

9.3.4 IXP-based Research

IXP-based improvements for wide area applications are attained by enabling packet header-based customization of an incoming data stream, thereby offering services such as software routing, network monitoring, etc. The Vera programmable software router developed at Princeton [98] enables applications to reconfigure the router functionality and implement application-specific routing decisions and firewalling policies. Similarly, the DiffServ IXP-server allows applications to specify the service classes differentiated by the server in an application-specific manner [64]. Both approaches rely solely on network-level information embedded in packet headers to perform the specific functionality. The same types of extensions can be implemented with Rx stream handlers in our model. Furthermore, we can base this functionality of application-level header, and perform content-based routing, firewalling, or service differentiation.

The Netbind system is closest to our work [20], in the sense that it is concerned with extending the packet's data path, except their focus is on routing functionality which requires access to IP headers only. The reconfigurability supported by Netbind is rather costly, and assumes stopping and restarting of the IXP microengines. We believe that by limiting the IXP's scope as a ANP, we can reserve some of its resources, i.e. microengines, to enable more efficient reconfiguration mechanism, without causing service interruption.

Another set of research developments targeting the IXP NPs, is the development of programming environments that will facilitate the analyses of performance implications

of various packet processing actions. The Shangri-La programming environment under development at UT Austin [57] is one such example. A component of this project focuses on analyses of the system requirements for enabling fine-grained adaptation in the IXP runtime, such as online reconfigurations of microengines' instruction store, or enabling/disabling system components upon request. While this work aims to detect the system requirements imposed by networking applications, application-level services using SPLITS can benefit from more flexible NP designs.

Other research efforts exploit the programmability of the IXP NPs to develop efficient security services, such as intrusion detection [26, 71, 24], resource discovery in peer-to-peer networks, or deliver services for a specific application domain, such as video-quality adjustments for video multicast [111] or quality provision [115, 19].

9.4 Programming Models

Finally, the SPLITS programming model is similar to other programming models that are used to represent computation in network-level services, such as the model adopted by the active networking community [46] and the programming model developed by Intel, specifically to express the development of communication-services on the IXP NPs [51, 54]. The active networking model represented through PLAN [46] and Intel's programming model that was developed specifically for the network processors used in our work are network-level models. The PLAN model does not prohibit higher level activities, these are constrained by the safety and security problems of the operational environment. Both programming models proposed by Intel, MicroACEs and microblocks, support only network-level activities, associated with the network-level headers of the packets that are handled. All activities are associated with all network-level packets on the data path, to form a single multi-stage pipeline. There are no distinct queues that enable separate flows, except for special exception cases. In Chapter 4 we established the SPLITS and these lower-level model. In addition, we mapped to the SPLITS model higher-level models used for streaming applications, such as Spidle and Streamit [104, 27], or for publish/subscribe systems such as ECho [30]. This demonstrated the generality of the SPLITS model, and further supports

our claim that a rich variety of services can be deployed on the host-ANP resources.

CHAPTER 10

CONCLUDING REMARKS

This chapter summarizes key conceptual and experimental contributions of this dissertation, and discusses some opportunities for future research directions.

10.1 Contribution

The main contributions of this research are the creation of SPLITS, a Software architecture for Programmable LIghtweight Stream handling, and its key abstraction - stream handler. SPLITS enables the joint use of standard hosts and their attached network processors and creates integrated host-ANP platforms. We explore the idea of using NPs closely tied to host nodes, thereby creating a computational platform that can deliver increased efficiency for variety of applications and services. The goals are to attain improvements in end-user application performance, to more efficiently utilize server capacity, and to offer new services at no or little additional performance overheads perceived by end users. SPLITS enables the dynamic creation of data paths through the host-ANP execution contexts, and the dynamic creation, deployment, and reconfiguration of application-specific network- and application-level processing on streaming data.

The stream handlers used by SPLITS represent lightweight, parameterizable, computational units. They can be applied at various points of the stream data path through the host-ANP platform, thereby enabling a rich set of application specific services. Stream handlers rely on the use binary format descriptors for accessing application-level data, which enables us to duplicate for packet bodies the elements that make it easy for NPs to perform header-based operations: known header formats, offsets, and types and sizes of fields. As a result, stream handlers can be integrated with the receive- or transmit-side protocol processing. Therefore, application-specific action can be executed with lower delays, unnecessary loads can be detected and prevented from affecting the system in a more timely manner,

and even data increasing services can be supported efficiently by avoiding repeated copying and protocol stack traversals. Furthermore, by enabling services to execute on the ANP, applications benefit from the optimized NP hardware, its efficient support for large data movement and built-in hardware parallelism, even for computationally intensive processing performed on application-level data in ANP memory.

We demonstrate that programmable network processors are highly capable of performing certain classes of application-specific processing on actual data content. However, the classes of services supported are limited due to the resource limitations on the NPs. The proposed software architecture integrates the host and the ANP resources, and creates a computational and communication platform that can deliver performance improvements to application services that can benefit from the NPs customized hardware, while still sustaining arbitrary services on the host's general purpose node. Furthermore, this architecture enables the dynamic deployment of application specific actions across host-ANP boundaries, and their reconfiguration, so as to best match the current platform resources and changes in application needs. Finally, we identify additional support tools required by SPLITS to better address the dynamic interactions required by the targeted applications.

10.2 Future Directions

This thesis opens several opportunities for future research directions. Some are directly connected to the present status of SPLITS, while others are related to the investigation of issues raised by SPLITS in other contexts.

Our short term research goals, include exploring the possibilities of implementing support services on top of SPLITS that will enhance middleware systems, such as those used in publish/subscribe systems, and using the mapping described in our work. Specifically, we are interested in evaluating the performance impact of using vertically integrated customizations, at host-, kernel-, and ANP-level, to the systems ability to maintain QoS requirements in the face of highly dynamic operating conditions. This also includes the development of a SPLITS implementation based on the next generation network processors.

Most of the open question raised deal above all with the dynamic reconfigurability

aspects of SPLITS. First, compiler solutions are needed for dynamic code generation, necessary for a truly general reconfigurable system. Such compiler tools can be extended with some profiling ability, so as to estimate the resource requirements and performance costs of the handler. In addition, tools for runtime generation of binary format descriptions for the target ANP architecture are needed, which can be accomplished from user-accessible commonly available data description standards like IDL or WSDL, Java classes, or well-formed XML-based data descriptions, such as XML schemas.

Finally, the safety and security functionality required by SPLITS, and programmable networking devices, in general, should be addressed before a commercially viable implementation of this system can be considered.

REFERENCES

- [1] ACHARYA, A., UYSAL, M., and SALTZ, J., “Active disks: programming model, algorithms and evaluation,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, (San Jose, CA), 1998.
- [2] Agere Systems, *The Case for a Classification Language. White Paper*, 2003.
- [3] Akamai Technologies, Inc., *The Business Internet: A Predictable Platform for Profitable E-Business. White Paper*, 2004. <http://www.akamai.com>.
- [4] ALEXANDER, D. S., ARBAUGH, W. A., HICKS, M. W., KAKKAR, P., KEROMYTIS, A. D., MOORE, J. T., GUNTER, C. A., NETTLES, S. M., and SMITH, J. M., “The SwitchWare active network architecture,” *IEEE Network Special Issue on Active and Controllable Networks*, vol. 12, no. 3, 1998.
- [5] ALEXANDER, D. S., MENAGE, P. B., KEROMYTIS, A. D., ARBAUGH, W. A., ANAGNOSTAKIS, K. G., and SMITH, J. M., “The Price of Safety in An Active Network,” *Journal of Communications and Networks (JCN), special issue on programmable switches and routers*, vol. 3, pp. 4–18, Mar. 2001.
- [6] ANDERSEN, D. G., BALAKRISHNAN, H., KAASHOEK, M. F., and MORRIS, R., “Resilient Overlay Networks,” in *Proc. of 17th Symposium of Operating Systems Principles (SOSP-17)*, (Banff, Canada), 2001.
- [7] APOSTOLOPOULOS, G., AUBESPIN, D., PERIS, V., PRADHAN, P., and SAHA, D., “Design, Implementation and Performance of a Content-Based Switch,” in *Proc. of IEEE INFOCOM 2000*, vol. 3, (Tel Aviv, Israel), pp. 1117–1126, Mar. 2000.
- [8] BERSHAD, B., SAVAGE, S., PARDYAK, P., SIRER, E. G., BECKER, D., FIUCZYNSKI, M., CHAMBERS, C., and EGGERS, S., “Extensibility, Safety and Performance in the SPIN Operating System,” in *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, (Copper Mountain, CO), 1995.
- [9] BHATTACHARJEE, B., AMMAR, M., ZEGURA, E., SHAH, V., and FEI, Z., “Application-Layer Anycasting,” in *Proceedings of INFOCOM’97*, (Kobe, Japan), 1997.
- [10] BHATTACHARJEE, S., CALVERT, K., and ZEGURA, E. W., “An architecture for active networking,” in *Proceedings of High Performance Networking (HPN)*, (White Plains, NY), 1997.
- [11] BHATTI, N. T., HILTUNEN, M. A., SCHLICHTING, R. D., and CHIU, W., “Coyote: A System for Constructing Fine-Grain Configurable Communication Services,” *ACM Transactions on Computer Systems*, vol. 16, pp. 321–366, Nov. 1998.

- [12] BHOEDJANG, R., RÜHL, T., and BAL, H., “Efficient Multicast on Myrinet Using Link-Level Flow Control,” in *Proceedings of International Conference on Parallel Processing*, (Minneapolis, MN), pp. 381–390, 1998.
- [13] BHOEDJANG, R., RÜHL, T., and BAL, H., “User Level Network Interface Protocols,” *IEEE Computer*, vol. 31, no. 11, pp. 53–60, 1998.
- [14] BOVA, T. and KRIVORUCHKA, T., “Reliable UDP Protocol - Internet Draft,” Feb. 1999. draft-ietf-sigtran-reliable-udp-00.txt.
- [15] BRAUN, F., LOCKWOOD, J., and WALDVOGEL, M., “Protocol wrappers for layered network packet processing in reconfigurable networks,” *IEEE Micro*, vol. 22, pp. 66–74, Jan./Feb. 2002.
- [16] BUSSE, I., BEFFNER, B., and SCHULZRINNE, H., “Dynamic QoS Control of Multimedia Applications Based on RTP,” *Computer Communications*, 1996.
- [17] BUSTAMANTE, F., EISENHAEUER, G., SCHWAN, K., and WIDENER, P., “Efficient Wire Formats for High Performance Computing,” in *Proc. of Supercomputing 2000*, (Dallas, TX), Nov. 2000.
- [18] BUSTAMANTE, F., EISENHAEUER, G., WIDENER, P., SCHWAN, K., and PU, C., “Active Streams: An Approach to Adaptive Distributed Systems,” in *Proc. of 8th Workshop HotOS-VIII*, (Elmau/Oberbayern, Germany), May 2001.
- [19] CALVERT, K., GRIFFION, J., and WEN, S., “Lightweight Network Support for Scalable End-to-End Services,” in *Proceedings of SIGCOMM’02*, (Pittsburg, PA), 2002.
- [20] CAMPBELL, A. T., CHOU, S., KOUNAVIS, M. E., STACHTOS, V. D., and VICENTE, J. B., “NetBind: A Binding Tool for Constructing Data Paths in Network Processor-based Routers,” in *Proc. of IEEE OPENARCH’02*, (New York City, NY), June 2002.
- [21] CARZAGINA, A., ROSENBLUM, D. S., and WOLF, A. L., “Design and Evaluation of a Wide-area Event Notification Service,” *ACM Transactions on Computer Systems*, vol. 19, no. 3, pp. 332–383, 2001.
- [22] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., and SINGH, A., “SplitStream: High-Bandwidth Multicast in Cooperative Environments,” in *Proc. of 18th Symposium of Operating Systems Principles (SOSP-18)*, (Bolton Landing, NY), 2003.
- [23] CHANDRA, P., CHU, Y.-H., FISHER, A., GAO, J., KOSAK, C., NG, T. E., STEENKISTE, P., TAKAHASHI, E., and ZHANG, H., “Darwin: Customizable Resource Management for Value-Added Network Services,” *IEEE Network*, vol. 15, no. 1, 2001.
- [24] CHARITAKIS, I., PNEVMATIKATOS, D., MARKATOS, E., and ANAGNOSTAKIS, K., “S2I: a Tool for Automatic Rule Match Compilation for the IXP Network Processor,” in *Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPEs 2003)*, (Vienna, Austria), 2003.
- [25] Chelsio Communications, *The Terminator Architecture. White Paper*, 2004.

- [26] CLARK, C., LEE, W., SCHIMMEL, D., CONTIS, D., KONE, M., and THOMAS, A., "A Hardware Platform for Network Intrusion Detection and Prevention," in *Proceedings of The 3rd Workshop on Network Processors and Applications (NP3)*, (Madrid, Spain), 2004.
- [27] CONSEL, C., HAMDI, H., REVEILLERE, L., LENIN SINGARAVELU, YU, H., and PU, C., "Spidle: A DSL Approach to Specifying Streaming Applications," in *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering*, (Erfurt, Germany), Sept. 2003.
- [28] DONGARRA, J. J., BUNCH, J. R., MOLER, C. B., and STEWART, G. W., *LINPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM Publications), 1979.
- [29] DRUSCHEL, P. and ROWSTRON, A., "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *Proc. of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, (Heidelberg, Germany), Nov. 2001.
- [30] EISENHAEUER, G., BUSTAMANTE, F., and SCHWAN, K., "Event Services for High Performance Computing," in *Proc. of Ninth High Performance Distributed Computing (HPDC-9)*, (Pittsburg, PA), Aug. 2000.
- [31] EXchip Technologies, *Network Processor Designs for Next-Generation Networking Equipment. White Paper*, 1999.
- [32] FEGHALI, W., BURRES, B., WOLRICH, G., and CARRIGAN, D., "Security: Adding Protection to the Network via the Network Processor," *Intel Technology Journal*, vol. 6, no. 3, 2002.
- [33] FELDMEIER, D. C., MCAULEY, A. J., SMITH, J., BAKIN, D., MARCUS, W., and RALEIGH, T., "Protocol Boosters," *IEEE JSAC, Special Issue on Protocol Architectures for 21st Century*, vol. 16, pp. 437-444, Apr. 1998.
- [34] FIUCZYNSKI, M. E., MARTIN, R. P., OWA, T., and BERSHAD, B. N., "SPINE - A Safe Programmable and Integrated Network Environment," in *Proc. of 8th ACM SIGOPS European Workshop*, (Sintra, Portugal), Sept. 1998.
- [35] FLINN, J., NARAYANAN, D., and SATYANARAYANAN, M., "Self-Tuned Remote Execution for Pervasive Computing," in *Proc of 8th IEEE HotOS Conference*, (Elmau/Oberbayern, Germany), 2001.
- [36] FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., and GAUTHIER, P., "Cluster-based Scalable Network Services," in *Proc of Sixteenth ACM Symposium on Operating System Principles*, 1999.
- [37] FOX, G., WU, W., UYAR, A., and BULUT, H., "Design and Implementation of an Audio/Video Collaboration System based on Publish/subscribe Middleware," in *Proceedings of CTS04*, (San Diego, CA), 2004.
- [38] FU, X., SHI, W., AKKERMAN, A., and KARAMCHETI, V., "CANS: Composable, Adaptive Network Services Infrastructure," in *USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2001.

- [39] GANEV, I., EISENHAUER, G., and SCHWAN, K., “Kernel Plugins: When a VM is Too Much,” in *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM’04)*, (San Jose, CA), May 2004.
- [40] GAVRILOVSKA, A., MACKENZIE, K., SCHWAN, K., and McDONALD, A., “Stream Handlers: Application-specific Message Services on Attached Network Processors,” in *Proc. of Hot Interconnects 10*, (Stanford, CA), Aug. 2002.
- [41] GAVRILOVSKA, A., SCHWAN, K., NORDSTROM, O., and SEIFU, H., “Network Processors as Building Blocks in Overlay Networks,” in *Proc. of Hot Interconnects 11*, (Stanford, CA), Aug. 2003.
- [42] GAVRILOVSKA, A., SCHWAN, K., and OLESON, V., “Adaptable Mirroring for Cluster Servers,” in *Proc. of 10th High Performance Distributed Systems*, (San Francisco, CA), Aug. 2001.
- [43] GAVRILOVSKA, A., SCHWAN, K., and OLESON, V., “Practical Approach for Zero Downtime in an Operational Information System,” in *Proc. of 22nd International Conference on Distributed Computing Systems (ICDCS’02)*, (Vienna, Austria), July 2002.
- [44] GIBBONS, P. B., KRAP, B., KE, Y., NATH, S., and SESHAN, S., “IrisNet: An Architecture for a World-Wide Sensor Web,” *IEEE Pervasive Computing*.
- [45] GILL, C. D., KUHN, F., LEVINE, D., SCHMIDT, D. C., DOERR, B. S., , and SCHANTZ, R. E., “Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems,” in *Proceedings of the 1st International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*, (Phoenix, Arizona), Nov. 1999.
- [46] HICKS, M., KAKKAR, P., MOORE, J. T., GUNTER, C. A., and NETTLES, S., “PLAN: A Packet Language for Active Networks,” in *Proceedings of the 3rd International Conference on Functional Programming (ICFP’98)*, (Baltimore, Maryland), 1998.
- [47] HILTUNEN, M. A., IMMANUEL, V., and SCHLICHTING, R. D., “Supporting Customized Failure Models for Distributed Services,” *Distributed System Engineering*, vol. 6, pp. 103–111, Dec. 1999.
- [48] HILTUNEN, M. A. and SCHLICHTING, R. D., “The Cactus Approach to Building Configurable Middleware Services,” in *Proc. of the Workshop on Dependable System Middleware and Group Communication (DSMGC 2000)*, (Nuremberg, Germany), Oct. 2000.
- [49] HUSAK, D. and GOHN, R., *Network Processor Programming Models: The Key to Achieving Faster Time-to-Market and Extending Product Life. White Paper*. C-Port. A Motorola Company, 2001.
- [50] Intel Corporation, *Intel Network Processor Family*. <http://developer.intel.com/design/network/products/npfamily/>.
- [51] Intel Corporation, *Intel IXA SDA ACE Programming Framework*, 2001.

- [52] Intel Corporation, *IXP1200: Software Development Kit 2.0*, 2002.
- [53] Intel Corporation, *Intel IXP2400 Network Processor: Flexible, High-Performance Solution for Access and Edge Applications. White Paper*, 2003.
- [54] Intel Corporation, *Introduction to the Auto-Partitioning Programming Model Accelerating Custom Application Development on Intel IXP2xxx Network Processors. White Paper*, Oct. 2003.
- [55] KEETON, K., PATTERSON, D. A., and HELLERSTEIN, J. M., "A case for intelligent disks (idisks)," *SIGMOD Record*, vol. 27, Sept. 1998.
- [56] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., and KAASHOEK, M. F., "The Click Modular Router," Aug. 2000.
- [57] KOUNAVIS, M., KUMAR, A., VIN, H. M., YAVATCKAR, R., and CAMPBELL, A., "Directions in Packet Classification for Network Processors," *Network Processors Design: Issues and Practice*, vol. 2, 2003.
- [58] KRISHNAMURTHY, R., YALAMANCHILI, S., SCHWAN, K., and WEST, R., "Architecture and Hardware for Scheduling Gigabit Packet Streams," in *Proc. of Hot Interconnects 10*, (Stanford, CA), Aug. 2002.
- [59] KRISHNAMURTHY, R., SCHWAN, K., and ROSU, M., "A Network Co-Processor-Based Approach to Scalable Media Streaming in Servers," in *Proc. of International Conference on Parallel Processing (ICPP)*, Aug. 2000.
- [60] KRISHNASWAMY, V., WALTHER, D., BHOLA, S., BOMMAIAH, E., RILEY, G., TOPOL, B., and AHAMAD, M., "Efficient Implementation of Java Remote Method Invocation (RMI)," in *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, (Santa Fe, NM), 1998.
- [61] KUMAR, R., WOLENETZ, M., AGARWALLA, B., SHIN, J., HUTTO, P., PAUL, A., and RAMACHANDRAN, U., "DFuse: A Framework for Distributed Data Fusion," in *Proceedings of ACM SenSys'03*, (Los Angeles, CA), Nov. 2003.
- [62] LEGEDZA, U., WETHERALL, D. J., , and GUTTAG, J., "Improving The Performance of Distributed Applications Using Active Networks," in *Proc. of IEEE INFOCOM'98*, (San Francisco, CA), Apr. 1998.
- [63] LIAO, C., MARTINOSI, M., and CLARK, D. W., "Performance Monitoring in a Myrinet-Connected Shrimp Cluster," in *ACM Sigmetrics Symposium on Parallel and Distributed Tools (SPDT)*, Aug. 1998.
- [64] LIN, Y.-D., LIN, Y.-N., YANG, S.-C., and LIN, Y.-S., "DiffServ over Network Processors: Implementation and Evaluation," in *Proc. of Hot Interconnects 10*, (Stanford, CA), Aug. 2002.
- [65] Linsys, *Linksys, Broadband and Wireless Networking*. <http://www.linksys.com>.
- [66] LIU, L., TANG, W., BUTTLER, D., and PU, C., "Information Monitoring on the Web: A Scalable Solution," *World Wide Web Journal*, vol. 5, no. 4, 2001.

- [67] LIU, X., KREITZ, C., VAN RENESSE, R., HICKEY, J., HAYDEN, M., BIRMAN, K., and CONSTABLE, R., "Building reliable, high-performance communication systems from components," in *Proc. of 17th ACM Symposium on Operating Systems Principles (SOSP'99)*, (Kiawah Island, SC), Dec. 1999.
- [68] MACKENZIE, K., SHI, W., McDONALD, A., and GANEV, I., "An Intel IXP1200-based Network Interface," in *Proceedings of the Workshop on Novel Uses of System Area Networks at HPCA (SAN-2 2003)*, (Anaheim, CA), Feb. 2003.
- [69] MOSBERGER, D. and PETERSON, L. L., "Making paths explicit in the Scout operating system," in *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, 1996.
- [70] OLESON, V., SCHWAN, K., EISENHAUER, G., PLALE, B., PU, C., and AMIN, D., "Operational Information Systems - An Example from the Airline Industry," in *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.
- [71] OTEY, M., NORONHA, R., PARTHASARATHY, S., and PANDA, D. K., "NIC-based Intrusion Detection: A Feasibility Study," in *Proceedings of the Workshop on Data Mining for Cyber Threat Analysis*, 2002.
- [72] PAI, V., COX, A., PAI, V., and ZWAENEPOEL, W., "A Flexible and Efficient Application Programming Interface (API) for a Customizable Proxy Cache," in *Proc. of 4th USENIX Symposium on Internet Technologies and Systems*, (Seattle, WA), 2003.
- [73] PALLICKARA, S. and FOX, G., "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids," in *Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003*, (Rio Janeiro, Brazil), 2003.
- [74] PARKER, M., DAVIS, A., and HSIEH, W., "Message-Passing for the 21st Century: Integrating User-Level Networks with SMT," in *Proc. of MTEAC 2001*, 2001.
- [75] Path 1 Network Technologies, *Professional Digital Video Gateways for the Broadcaster and Multi-Service Operator: Delivered by Path 1 Network Technologies* and Intel Network Processors. White Paper*, 2002. <http://www.intel.com/design/network/casestudies/path1.htm>.
- [76] PETERSON, L. L., KARLIN, S. C., and LI, K., "OS support for general-purpose routers," in *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.
- [77] PLALE, B., ELLING, V., EISENHAUER, G., SCHWAN, K., KING, D., and MARTIN, V., "Realizing Distributed Computational Laboratories," *International Journal of Parallel and Distributed Systems and Networks*, vol. 2, no. 3, 1999.
- [78] POELLABAUER, C., ABASSI, H., and SCHWAN, K., "Cooperative Run-time Management of Adaptive Applications and Distributed Resources," in *Proc. of the 10th ACM Multimedia Conference*, (Juan-les-Pins, France), Dec. 2002.
- [79] POELLABAUER, C., SCHWAN, K., AGARWALA, S., GAVRILOVSKA, A., EISENHAUER, G., PANDE, S., PU, C., and WOLF, M., "Service Morphing: Integrated System-

- and Application-Level Service Adaptation in Autonomic Systems,” in *Proc. of the 5th Annual International Workshop on Active Middleware Services (AMS 2003)*, (Seattle, WA), June 2003.
- [80] PONNEKANTI, S. R. and FOX, A., “SWORD: A Developer Toolkit for Web Service Composition,” in *Proc. of the 11th World Wide Web Conference*, (Honolulu, HI), 2002.
- [81] PU, C., AUTREY, T., BLACK, A., CONSEL, C., COWAN, C., INOUE, J., KETHANA, L., WALPOLE, J., and ZHANG, K., “Optimistic Incremental Specialization: Streamlining a Commercial Operating System,” in *Proc. of the 15th ACM Symposium on Operating Systems Principles (SOSP’95)*, (Copper Mountain, Colorado), Dec. 1995.
- [82] PU, C., SCHWAN, K., and WALPOLE, J., “Infosphere Project: System Support for Information Flow Applications,” *ACM SIGMOD Record*, vol. 30, Mar. 2001.
- [83] RadiSys Corporation, *Radisys ENP-2611 Data Sheet*. http://www.radisys.com/files/ENP-2611.07-1236-02_0803.pdf.
- [84] RAMAN, B. and KATZ, R., “An Architecture for Highly Available Wide-Area Service Composition,” *Computer Communications Journal, special issue on “Recent Advances in Communication Networking”*, May 2003.
- [85] REGNIER, G., MINTURN, D., MCALPINE, G., SALETTORE, V., and FOONG, A., “ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine,” in *Proc. of Symposium of Hot Interconnects*, (Stanford, CA), 2003.
- [86] RENAMBOT, L., BAL, H., GERMANS, D., and SPOELDER, H., “CAVEStudy: an Infrastructure for Computational Steering in Virtual Reality Environments,” in *Proc. of High Performance Distributed Computing (HPDC-9)*, (Pittsburgh, PA), Aug. 2000.
- [87] ROSU, M. C. and ROSU, D., “An Evaluation of TCP Splice Benefits in Web Proxy Servers,” in *Proc. of WWW2002: The 11th International World Wide Web Conference*, (Honolulu, Hawaii), 2002.
- [88] ROSU, M.-C., SCHWAN, K., and FUJIMOTO, R., “Supporting Parallel Applications on Clusters of Workstations: The Virtual Communication Machine-based Architecture,” *Cluster Computing, Special Issue on High Performance Distributed Computing*, May 1998.
- [89] ROWSTRON, A., KERMARREC, A.-M., CASTRO, M., and DRUSCHEL, P., “SCRIBE: The design of a large-scale event notification infrastructure,” in *Proc of 3rd International Workshop on Networked Group Communication*, (London, UK), 2001.
- [90] ROY, S., ANKCORN, J., and WEE, S., “An Architecture for Componentized, Network-Based Media Services,” in *Proc. of IEEE International Conference on Multimedia and Expo*, July 2003.
- [91] SAITO, Y., BERSHAD, B., and LEVY, H., “Manageability, Availability, and Performance in Porcupine: A Highly Scalable Cluster-based Mail Service,” in *Proc of 17th ACM SOSP, OS Review*, (Kiawah Island Resort, SC), Dec. 1999.

- [92] SANDERS, M., KEATON, M., BHATTACHARJEE, S., CALVERT, K., ZABELE, S., and ZEGURA, E., "Active Reliable Multicast on CANEs: A Case Study," in *Proc. of IEEE OpenArch 2001*, (Anchorage, Alaska), Apr. 2001.
- [93] SENAPATHI, S., CHANDRASEKHARAN, B., STREDNEY, D., SHEN, H.-W., and PANDA, D. K., "QoS-aware Middleware for Cluster-based Servers to Support Interactive and Resource-Adaptive Applications," in *Proceedings of the 12th Symposium on High Performance Distributed Computing (HPDC-12)*, (Seattle, WA), 2003.
- [94] SHAH, N. and KEUTZER, K., "Network Processors: Origin of Species," in *Proceedings of the 17th International Symposium on Computer and Information Science (ISCIS XVII)*, 2002.
- [95] SiTera Corporation, *PRISM IQ2000. Product Brief*, 2000.
- [96] SMITH, J., HADZIC, I., and MARCUS, W., "ACTIVE Interconnects: Let's Have Some Guts," in *Proc. of Hot Interconnects 6*, (Palo Alto, CA), Aug. 1998.
- [97] SNOEREN, A. C., CONLEY, K., and GIFFORD, D. K., "Mesh Based Content Routing using XML," in *Proc. of 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, (Banff, Canada), Oct. 2001.
- [98] SPALINK, T., KARLIN, S., PETERSON, L., and GOTTLIEB, Y., "Building a Robust Software-Based Router Using Network Processors," in *Proc. of 18th SOSP'01*, (Chateau Lake Louise, Banff, Canada), Oct. 2001.
- [99] STALLINGS, W., "IPv6: The New Internet Protocol," *IEEE Communications Magazine*, 1996.
- [100] STEENKISTE, P., CHANDRA, P., GAO, J., and SHAH, U., "An Active Networking Approach to Service Customization," in *DARPA Active Networks Conference and Exposition (DANCE)*, pp. 305–318, IEEE Computer Society, 2002.
- [101] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., and BALAKRISHNAN, H., "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications," in *Proc. of ACM SIGCOMM 2001*, (San Diego, CA), Aug. 2001.
- [102] TAYLOR, D. E., LOCKWOOD, J. W., SPROULL, T. S., TURNER, J. S., and PARLOUR, D. B., "Scalable IP Lookup for Programmable Routers," in *Proc. of IEEE Infocom 2002*, (New York, NY), June 2002.
- [103] TAYLOR, D. E., TURNER, J. S., and LOCKWOOD, J. W., "Dynamic Hardware Plugins (DHP): Exploiting Reconfigurable Hardware for High-Performance Programmable Routers," in *Proc. of 4th IEEE Conference on Open Architectures and Network Programming (OPENARCH'01)*, (Anchorage, AK), Apr. 2001.
- [104] THIES, W., KARCZMAREK, M., and AMARASINGHE, S., "StreamIt: A Language for Streaming Applications," in *International Conference on Compiler Construction (ICCC'02)*, (Grenoble, France), Apr. 2002.
- [105] Tibco Software Inc., *Tibco ActiveEnterprise: XML Tools*. <http://www.tibco.com/solutions/products/extensibility/>.

- [106] WETHERALL, D. J., “Active Network Vision and Reality: Lessons from a Capsule-based System,” in *Proc. of the 17th ACM Symposium on Operating System Principles (SOSP’99)*, (Kiawah Island, SC), Dec. 1999.
- [107] WHEELER, B. and GWENNAP, L., *The Guide to Network Processors*. The Linley Group, 2003.
- [108] WIDENER, P., EISENHAEUER, G., SCHWAN, K., and BUSTAMANTE, F., “Open Metadata Formats: Efficient XML-Based Communication for High Performance Computing,” *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, no. 5, pp. 315–324, 2002.
- [109] WOLF, M., CAI, Z., HUANG, W., and SCHWAN, K., “Smart Pointers: Personalized Scientific Data Portals in Your Hand,” in *Proc. of Supercomputing 2002*, Nov. 2002.
- [110] XIE, Y., O’HALLARON, D., and REITER, M., “A Secure Distributed Search System,” in *Proc. of 11th Symposium on High Performance Distributed Systems (HPDC-11)*, (Edinburgh, Scotland), 2002.
- [111] YAMADA, T., WAKAMIYA, N., MURATA, M., and MIYAHARA, H., “Implementation and evaluation of video-quality adjustment for heterogeneous video multicast,” in *Proceedings of The 8th Asia-Pacific Conference on Communications*, 2002.
- [112] YOCUM, K. and CHASE, J., “Payload Caching: High-Speed Data Forwarding for Network Intermediaries,” in *Proc. of USENIX Technical Conference (USENIX’01)*, (Boston, Massachusetts), June 2001.
- [113] ZHAO, Y. and STORM, R., “Exploiting Event Stream Interpretation in Publish-Subscribe Systems,” in *Proc. of ACM Symposium on Principles of Distributed Computing*, (Newport, RI), Aug. 2001.
- [114] ZHOU, D., SCHWAN, K., EISENHAEUER, G., and CHEN, Y., “JECho – Interactive High Performance Computing with Java Event Channels,” in *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS’01)*, Apr. 2001.
- [115] ZHOU, W., LIN, C., LI, Y., and TAN, Z., “Queue Management for QoS Provision Build on Network Processor,” in *Proceedings of the 9th Workshop on Future Trends in Distributed Computing Systems*, (San Juan, Puerto Rico), 2003.
- [116] ZHUANG, X., SHI, W., PAUL, I., and SCHWAN, K., “Efficient Implementation of the DWCS Algorithm on High-Speed Programmable Network Processors,” in *Proc. of Multimedia Networks and Systems (MMNS)*, Oct. 2002.

VITA

Ada Gavrilovska was born in Skopje, Macedonia on May 25, 1975. She received her BS in Electrical and Computer Engineering from the University Sts. Cyril and Methodius, Skopje in 1998. That summer she joined the Systems Group at the College of Computing at Georgia Institute of Technology. After obtaining her MS degree in Computer Science in December 1999, she remained at Georgia Tech, and enrolled in the Ph.D. program.

In July 2004, under the supervision of Prof. Karsten Schwan, Ada completed her Ph.D. dissertation entitled “SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors”. Her research interest range from distributed systems, to active middleware, to extensible network infrastructures and active networking.