The Dissertation Committee for Ravindranath Kokku
certifies that this is the approved version of the following dissertation:

ShaRE: Run-time System for High-performance Virtualized Routers

Committee:

Harrick M. Vin, Supervisor

Lorenzo Alvisi

Michael D. Dahlin

Charles G. Plaxton

Ramakrishnan Rajamony

Raj Yavatkar

ShaRE: Run-time System for High-performance Virtualized Routers

by

Ravindranath Kokku, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2005

To My Parents

Acknowledgments

I believe that the process of earning a PhD degree fundamentally changes the way one thinks. And one's advisor is the most significant contributor to such a change. Harrick Vin has striven hard to make me think differently, to convert me into a scientist from an engineer. Harrick's insistence on elegance of presentation both in writing and in talking has been quite valuable in honing my skills. His rigorous understanding, cute insights and passionate criticism have made my dissertation better by the day, and my PhD a pleasant experience overall. I have cherished many incredibly lengthy and interesting, but never tiring, meetings with him on both technical and philosophical issues. His personal warmth and support during happy and tough times, and his patience and nicety during heated discussions and everyday interactions have given me the necessary protection and confidence to keep going. Thanks for everything Harrick.

Over the past five years, I have also been fortunate to work closely with Lorenzo Alvisi and Mike Dahlin. Both have been great mentors in their own right. I am thankful to Lorenzo for believing in me more than I did in myself at one point and introducing me to LASR, for giving me opportunities to develop professionally, for chiding and guiding me every time he thought I was losing my track, and for making me smile, laugh and feel good with his witty remarks, quick jokes and short stories. In short, with Lorenzo around, life is beautiful everyday.

Mike is one of the most fascinating researchers I have ever interacted with.

I have always been awe-struck by his ability to think multiple levels ahead of me and expand or replace my ideas with his ideas (many of which were lot more elegant and cleaner) within minutes of starting a discussion with him. His constant interest and guidance to give the best shape to my work, his principle of leading-by-example rather than forcing a particular methodology on students, his wisdom for developing useful systems, his stress on simplicity and clarity of solutions, his demonstration of next-to-God coding skills to teach us not to give up till the last minute of a paper deadline, are but a few nuggets that have left a lasting impression on me.

I am grateful to my other committee members Greg Plaxton, Ram Rajamony, and Raj Yavatkar for their insightful comments and questions that shaped my dissertation at various stages. I am thankful to Ram Rajamony for his interest in my professional life and suggestions during the last five years, and for two pleasant internships at IBM Austin Research Lab.

My collaborators Vinod Balakrishnan, Taewon Cho, Aaron Kunze, Ajay Mahimkar, Jayaram Mudigonda, Ram Rajamony, Taylor Riche, Nishit Shah, Sadia Sharif, Upendra Shevade, Arun Venkataramani and Praveen Yalagandula have contributed in different capacities to my success. Also, many thanks to all LASRians for discussions at EasyMondays, and comments on paper drafts and practice talks.

I am thankful to Sara Strandtman, Gloria Ramirez, and Katherine Utz for providing outstanding administrative support and for patiently fixing the unique problems I took to them; without them my life would have been hard.

I am glad to have had roommates like Ravi Chamarajnagar, Sam Idicula, Vishv Jeet, Rama Kotla and Arun Venkataramani, and other friends like Aseem Bajaj, Kedarnath Balakrishnan, Bharat Chandra, Mini Dwivedi, Thomas George, Subbu Iyer, Prem Melville, Srujana Merugu, Joseph Modayil, Aniket Murarka, Amol Nayate, Alison Norman, Dibyendu Sengupta and Praveen Yalagandula who joined me in celebrating during happy times, and supported me through tough times.

Several of them have also served as sounding boards for many of my ideas. Amol, Praveen, Prem, Rama and Srujana, especially, have always been there for me when I needed help, care, or just to feel good. Thanks to my coffee-buddies Amol, Aniket, Arun, Jeff, Jiandan, Joseph, Kotla, Lei, Misha, Prem and Taylor who gave me a reason to go to school even when I did not feel like working—to exchange gossip at Starbucks or Peet's.

Rachitha, Rachana, Radhika and Raghu Jadala have been entertaining me in more ways than I imagined ever since I met them. Every minute I spend playing with Rachitha and Rachana brings out the child in me and makes me forget all the tensions I have.

I am indebted to *Asha for Education*—an organization for socially and economically uplifting underprivileged children in India through education—for giving me reasons to get out of a crisis I went through in the middle of my PhD. Thanks to Asha, I had a chance to be involved in building a school in one of the poorest and underdeveloped villages of India, an experience I would cherish forever.

My brother Kishore has been my best friend, always.

Finally, the most important people in my life—my parents—have set lasting examples of not giving up even during hardest of times. Their life of struggles inspires me constantly to hold on to the belief that hardwork pays-off ultimately. This dissertation is dedicated to them.

Ravindranath Kokku

The University of Texas at Austin
December 2005

ShaRE: Run-time System for High-performance Virtualized Routers

Publication No. _____

Ravindranath Kokku, Ph.D.
The University of Texas at Austin, 2005

Supervisor: Harrick M. Vin

The enormous size and scope of today's Internet make the deployment of new network technologies and advanced services difficult. One strategy to address such ossification is to combine two evolving trends in building network services—overlay networks and network virtualization. The combination allows multiple overlay networks providing alternate end-to-end services to co-exist on a shared virtualized substrate. The building blocks of such a substrate—termed *virtualized routers*—utilize programmable multi-processor hardware to provide the flexibility of hosting diverse services simultaneously, and process packets at high rates.

In this dissertation, we describe the design and implementation of ShaRE, the first run-time system for such *high-performance, multi-service virtualized routers*. ShaRE provides two functionalities. First, ShaRE multiplexes hardware resources among multiple services while preventing damaging interactions and instabilities. Second, ShaRE exposes a packet-processing-oriented resource abstraction layer and

includes a set of adaptation mechanisms to simplify the programmability and portability of high-performance network services.

Virtualized routers exhibit several unique characteristics that make the above functionalities challenging. First, network services are required to process packets within a small delay from their arrivals. Second, at these time-scales, network traffic is bursty and often hard-to-predict. Third, the overhead of reallocating processors from one service to another in current router hardware is larger than the packet processing time. Finally, to achieve high performance, router hardware (unlike conventional systems) includes a complex array of resources such as multi-core processors, fixed-function co-processors, and inter-processor communication mechanisms.

In designing ShaRE, we make three contributions. First, we design Everest, an *agile* and *wary* scheduler for allocating processors to services in high-performance routers. Everest's agility and wariness help cope with stringent delay tolerance, and difficult-to-predict and significant fluctuations in packet arrival rates, and large context-switch overheads. Second, we demonstrate that the efficacy of a scheduler depends on various system, workload and service characteristics, and design variants of Everest that perform the best under different characteristics. Finally, to minimize the overhead of replicating services or check-pointing and migrating services from one processor to another, we identify and exploit several unique characteristics of network services. We instantiate our contributions on Intel's IXP2400 programmable router hardware.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation describes the design and implementation of ShaRE (the Shangri-La Run-time Environment)—the first run-time system for the emerging class of *high-performance, multi-service virtualized routers* [13, 17, 83].

1.1 High-performance Virtualized Routers

Over the past decade, new uses and commercialization have pushed the Internet well beyond the expectations of its designers. What started off as a simple network of tens of nodes in the 1970s, has now grown into a complex network of thousands of networks, with millions of nodes, with thousands of ISPs [17, 22]. This enormous scale and popularity has led to a shift in the requirements of today's Internet; the following examples illustrate this shift [33].

- *Performance:* Unlike the initial Internet applications that were simple and best-effort in nature (e.g., email, ftp), today's applications are quite complex, and often require the network to provide end-to-end performance guarantees to users (e.g., e-commerce, search, video streaming).

- *Ease of Use:* While the initial Internet was designed and used by technologists, today's Internet includes substantial number of unsophisticated users. Further, the devices that the users use to connect to the Internet have evolved from PCs to diverse devices such as Personal Digital Assistants, cellphones, sensor devices, etc. Such a trend advocates the requirement of external support (from some entity *in* the Internet) to simplify the configuration and control of the devices users own.

- *Protection:* The lack of accountability (of who can send what into the network) has led to proliferation of unwanted and malicious traffic (e.g., spam, trojans, viruses, worms). In today's complex network that supports diverse applications for a large base of unsophisticated users, leaving the protection responsibility to the end users is ineffective and often dangerous; ideally the network to which a user connects should protect him (and his devices) from unwanted traffic.

- *Anonymity:* Today's users desire anonymity of their activity on the Internet. Examples of circumstances that require anonymity (to avoid others from learning a user's behavior) include online purchase of certain items, visiting certain Web sites, freedom of expression about controversial issues, and online voting. However, the Internet currently exposes one's identity in several ways such as IP addresses and email addresses.

Despite the growing need for richer functionality to cope with the changing requirements, the network architecture and services provided by the Internet have changed little since its invention. The introduction and adoption of new network architectures and advanced services not only requires modifications to routers and host software, but also requires agreement among multiple ISPs. The difficulty in reaching consensus and the substantial costs of upgrading the infrastructure have

hindered the evolution of the Internet—this phenomenon is termed Internet Ossification [13, 17, 82, 83].

It has been argued [13, 17, 83] that two recent trends—*overlay networks* and *virtualization*—can be combined to address the Internet Ossification effectively and systematically. Over the past few years, overlay networks have often been used to augment the current Internet [1, 26, 49, 65, 81, 95, 96, 105, 109]; content distribution networks (CDNs) are perhaps the most prominent examples of such overlays. Because overlay networks can be accessed by a user through proxies, and the decision about whether or not to use an overlay network can be made on a per-user/application basis, overlay networks that provide new services (e.g., caching/prefetching, multi-path routing, etc.) have been effective in attracting significant amount of real user traffic.

Virtualization, on the other hand, utilizes a shared substrate of nodes (termed virtualized routers) connected by physical links (either directly or through other routers) to support many logical routers and logical links (Figure 1.1). Each logical router supports a different network architecture or service [13]. An overlay network can now be created by a combination of logical routers connected by logical links. This allows the cost and the overhead of creating and managing the overlay to be amortized across multiple architectures and services. Thus, virtualization offers an effective mechanism for introducing, experimenting with, and deploying (for real use) new architectures and services in the Internet. The success of PlanetLab is indicative of the effectiveness of this paradigm [31, 82, 83].

Today, virtualized routers are built using commodity PCs [31, 82, 83]; consequently, they can only support relatively low traffic levels. While such a setup is sufficient for validating the effectiveness of new approaches, it is inadequate for evaluating their scalability or deploying them in realistic high-bandwidth environments; PCs based on traditional general purpose processors do not provide sufficient pro-

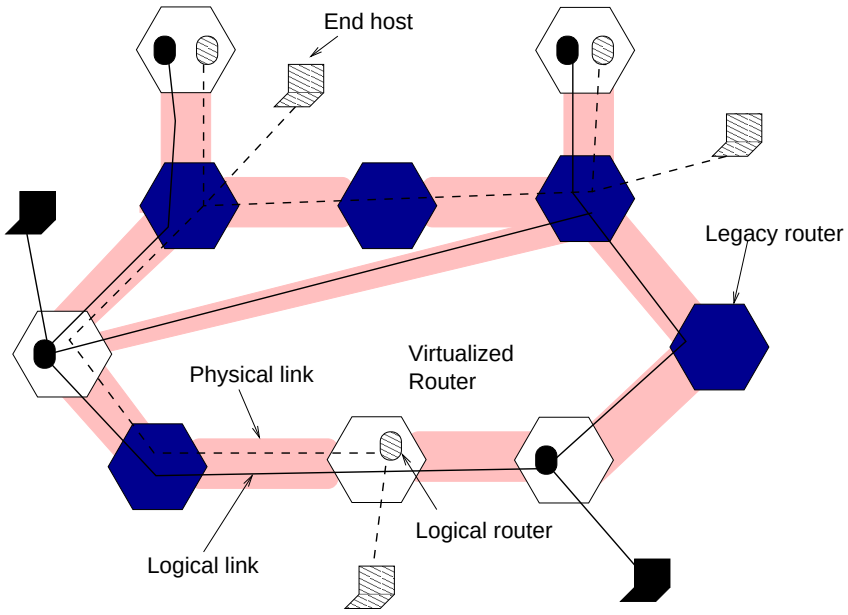


Figure 1.1: High-performance virtualized routers in the Internet. White hexagons represent virtualized routers that host multiple logical routers to provide advanced services. For instance, the solid ovals connected by solid lines form an overlay network providing a particular service. Similarly the striped ovals connected by dotted lines form another overlay network. End hosts can either connect to one of the virtualized routers to receive the richer services hosted by the virtualized router, or use the legacy routers to receive traditional services (like IP forwarding). Virtualized routers can either replace a legacy router at a location completely (like the bottom white hexagons) or supplement the legacy router (like the top white hexagons) by providing richer services to only a part of the traffic. Figure derived from [13].

cessing capacity to handle high-bandwidth links. Since in the networking domain multiple packets can be processed simultaneously, a natural solution to achieve high-performance is to employ parallel resources.

While using a cluster of PCs connected by a COTS (commodity-off-the-shelf) switch is an alternative to support the necessary scalability, a more cost-effective solution is to utilize the emerging multi-core multi-threaded processors [5, 12, 15, 19, 20, 57], integrated with other fixed-function logic (such as hash units, crypto units,

and TCAMs). For instance, Intel®'s IXP2400 network processor [57] supports 8 RISC cores (called micro-engines or MEs) each with 8 hardware threads. Each ME has a private instruction store of size 4K instructions, 640 words of private memory, 256 general-purpose registers, 128 next-neighbor transfer registers, and 512 SRAM and DRAM data transfer registers. MEs share a 16KB on-ship scratchpad as well as off-chip SRAM (upto 128MB) and DRAM (upto 1GB). MEs can communicate via next-neighbor registers or via shared memory. IXP2400 also includes a fixed-function hash unit, and a general-purpose XScale® core.

Although the necessary hardware is available to support high-performance virtualized routers, the tools available for programming such hardware are still in a nascent stage [5, 14, 21, 24, 50, 70, 87, 91, 102]. They provide some high-level primitives to avoid programming in assembly, but expose the hardware features to the programmers for them to make informed decisions about resource allocation. The lack of right abstractions (due to the exposed hardware) makes network services unportable across different versions and instantiations of the hardware, and hurts the flexibility to best-utilize the hardware resources based on service deployment scenarios; programmers have to determine resource allocations statically at design time, which can be complex, tedious and error-prone.

1.2 The Shangri-La Runtime Environment

The purpose of the Shangri-La Runtime Environment (ShaRE) is to simplify the development and deployment of innovative Internet services on virtualized routers built using the emerging multi-core multi-threaded hardware. ShaRE provides two key pieces for deploying large number of overlays on substrates built using such hardware. First, ShaRE multiplexes resources among multiple services in order to avoid the need to deploy dedicated hardware for each service while preventing damaging interactions and instabilities among services. Second, ShaRE exposes a

packet-processing-oriented resource abstraction layer (pRAL) that hides the details of the resources from the programmer to simplify programmability and improve portability. Underneath pRAL, ShaRE includes a set of adaptation mechanisms that allow services to make the best use of hardware resources to achieve high-performance.

When combined with overlay networks and programmable router hardware, ShaRE enables creation of commercial-scale overlay hosting facilities (much like today's web hosting facilities) with high-performance virtualized routers and dedicated links. Such a *programmable network substrate* greatly lowers the barrier for introducing new network architectures and services into the Internet.

1.2.1 Challenges

Although the requirements of multiplexing resources efficiently and improving programmability/portability are traditional operating system tasks, the domain of virtualized routers introduces several challenges that render conventional OS solutions to these problems ineffective.

- The arrival rate of packets at virtualized routers is at least two to three orders of magnitude greater than the arrival rate of requests for other hosting environments (e.g., millions of packets per seconds vs. thousands of requests per second in a web hosting environment). Further, network traffic is bursty and hard-to-predict, especially at fine timescales (upto 100s of milliseconds) [84, 111].

- The *context-switch overhead* of reallocating processing resources from one service to another in current hardware is large and is often two to three orders of magnitude greater than the time required to process a packet. For instance, on Intel®'s IXP2400 network processor, loading the instruction store of a *micro-engine* with the code for a new service incurs a context-switch time

that is greater than $1ms$ (see Chapter 4), while a packet can often be processed within tens of microseconds.

- Network services are required to process packets within a small delay from their arrivals. In contrast to conventional operating systems and hosting environments where a relatively large delay (e.g., $10ms$) is tolerable even for interactive applications, in this environment adding even a $1ms$ scheduling delay per hop (router) for each packet is prohibitive.

- To support high-bandwidth networks and complex packet processing services, programmable routers include a complex array of resources including multiple many-core processors; multiple threads per core; configurable processors; fixed-function co-processors for hashing, crypto, associative lookup, or other functions; complex memory hierarchies; and multiple inter-processor communication mechanisms [57]. Further, the router architecture evolves with time to incorporate advanced features into the hardware to speed-up packet processing.

1.2.2 Contributions

To address each of these challenges, we introduce three innovations.

- We design Everest, an *agile* and *wary* scheduler for allocating processing resources to network services in high performance routers built on state-of-the-art programmable hardware. Everest's *agility* allows us to cope with stringent delay tolerance and difficult-to-predict and significant fluctuations in packet arrival rates at small timescales. Everest's *wariness* copes with the relatively large context switch overheads; it does not context switch a processor unless necessary. As a result, Everest is a variable-quantum, non-work-conserving scheduler. Our experiments demonstrate that, as compared to conventional

schedulers, Everest reduces by more than an order of magnitude the average delay experienced by packets as well as the number of packets that experience delay greater than the desired threshold.

- Realizing that Everest is only a point-solution suitable for today's virtualized routers, we perform a design space exploration of different schedulers under various system, service and workload characteristics. We demonstrate that the efficacy of a scheduler depends on the relationship between various characteristics, and design variants of Everest that perform the best under different characteristics.

- We identify and exploit several unique characteristics of network services to minimize the overhead of replicating services dynamically or check-pointing and migrating services from one processor to another in response to workload fluctuations. To facilitate such adaptation and to simplify programmability, ShaRE provides pRAL that exports abstract interfaces for each type of hardware resource (e.g., processors, inter-processor communication and synchronization mechanisms), and provides diverse implementations for each resource type. ShaRE dynamically binds the most-appropriate implementation to the interface as available resources and workload requirements change. Such abstraction hides specific hardware details from the programmers and compilers and yet allows services to utilize available resources effectively to maximize performance. We describe a prototype of ShaRE built on an Intel®'s IXP2400 network processor. We demonstrate the effectiveness of ShaRE by designing a high-performance virtualized router that hosts several services, such as a port-scan detector, a TCP-SYN flood detector, and a worm signature matcher.

In summary, ShaRE advances the state-of-the-art in designing multi-service routers in two significant ways. First, ShaRE simplifies the programming of complex,

multi-threaded multi-processor architectures, and thereby facilitates rapid development and deployment of high-performance network services. Second, ShaRE multiplexes available processing resources among competing services efficiently; this makes the system robust against traffic fluctuations and attacks, and reduces the overall processor provisioning requirements.

1.3 Outline

The rest of the dissertation is organized as follows. In Chapter 2, we describe the overall architecture of the Shangri-La programming environment for high-performance routers, and discuss the interaction of ShaRE with other components of the environment. In Chapter 3, we design and evaluate Everest, an agile and wary scheduler for high-performance virtualized routers built on today's hardware. We also perform a design space exploration of several schedulers under different system, service and workload characteristics, and design variants of Everest that perform the best for different characteristics. Chapter 4 presents the design of the resource abstraction layer along with the resource-binding and check-pointing mechanisms and their implementation on a platform containing Intel®'s IXP2400 network processor. In Chapter 5, we evaluate the benefits of ShaRE in developing several network services that form building blocks of high-performance virtualized routers. Chapter 6 summarizes our contributions and discusses future work.

Chapter 2

The Programming Environment

This chapter provides a brief overview of different components of the Shangri-La programming environment, and describes the interaction of ShaRE with the other components of the environment. Figure 2.1 illustrates the high-level architecture of the programming environment. The goal of the environment is to make programming network services on high-performance routers as easy as programming applications on today's workstations and servers. To realize this goal, Shangri-La includes Baker—a domain-specific language, a pipeline compiler [42], and ShaRE.

2.1 Baker

Network services receive, process, and transmit packets. These services are described using data-flow graphs of *packet processing functions (PPFs)*—each PPF is a well-defined set of operations on a packet. On processing a packet, some functions generate multiple output packets (e.g., multicast routing); some functions consume multiple packets and generate one output packet (e.g., IP de-fragmentation); while some other functions generate output packets without consuming any packets (e.g., keep alive). Many packet-processing functions are stateful; the state is accessed and

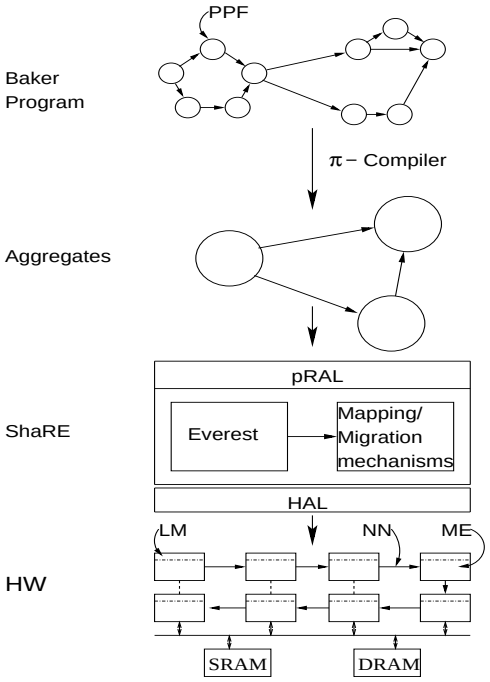


Figure 2.1: High level system architecture.

updated while processing packets belonging to a flow. A flow refers to a sequence of related packets (e.g., all packets to a destination, or all packets exchanged between two application end-points). The execution of PPFs is triggered by packet arrivals, timers, or other events (e.g., link failure). The sequence of functions executed for a packet depends on the packet's type (determined based on its header and content) and the state of the system.

Baker, a modular, domain-specific programming language, facilitates the development of such network services. Baker is syntactically similar to C, and—much like Click [67], Genesis [36], NetScript [110], NetBind [70], VERA [63], Scout [77], RouterPlugins [46], and PromethOS [64]—allows services to be specified as graphs of PPFs. Baker differs from past systems in two ways. First, it allows programmers to annotate fine-grained PPFs with information that enables efficient compilation of Baker programs onto complex, multi-processor architectures. Second, Baker exposes

to programmers the pRAL interfaces exported by ShaRE; this allows services to utilize available hardware resources effectively without exposing to the programmers any of the hardware details.

2.2 π-compiler

The *pi*peline compiler (π-compiler) addresses the question: *how should a network service utilize the parallel processing resources available in the router such that the packet processing throughput is maximized?* Most network services exhibit a high-degree of data-parallelism; there is often little dependence between packets belonging to different flows. Only the maintenance of shared per-flow state and other ordering constraints serialize the processing of packets belonging to a flow. Baker exposes and the π-compiler exploits this fact. In particular, given a service specified as a graph of PPFs [56, 67], the π-compiler decides how many *aggregates* to create. At one extreme, all the PPFs can be combined into a single *aggregate*; the run-time system replicates such an aggregate onto the multiple processors to process packets in parallel. In this case, each packet is processed completely at one processor. Alternately, the π-compiler can group PPFs from the service graph into a sequence of one or more pipeline stages [56, 71, 106] or *aggregates*, each of which can be mapped onto one or more processors. In this case, each packet is processed by multiple processors. Because of the streaming nature of processing performed by each aggregate (aggregates process packets continuously in a *loop*), to achieve high performance, it is crucial that code for the entire aggregate can fit into the instruction cache of a processor[1]. The construction of aggregates in π-compiler is driven by this requirement, along with considerations for minimizing packet communication and data sharing overheads.

[1]On some platforms, such as Intel®'s IXP2400 network processor that we use in our prototype, processors support only an instruction *store* and not a cache. In such cases, ensuring that each aggregate can fit completely within an instruction store is a hard requirement.

The aggregates constructed by the π-compiler are passed onto the code generator, which performs both machine dependent and machine independent optimizations to maximize the throughput sustained by the aggregate. For each aggregate, it produces loadable binaries for each type of processors available in the router.

2.3 ShaRE: Shangri-La Run-time Environment

ShaRE loads and runs these aggregates onto a multi-processor router. ShaRE includes both mechanisms and policies for mapping aggregates onto hardware resources. In particular, ShaRE exports the pRAL interface, supports multiple implementations of each pRAL interface, and includes a mechanism for binding pRAL interfaces to implementations dynamically. ShaRE also includes mechanisms to *replicate*, *checkpoint* and *migrate* aggregates to enable adaptation of resource allocations to aggregates dynamically. ShaRE performs adaptation of resource allocations at various timescales—at service activation time and at run-time. Adaptation at various timescales is necessary in a virtualized router; while adapting at service activation time is necessary to accommodate new services in the system, run-time adaptation is necessary to reduce the cumulative processing resources through statistical multiplexing, and also to provide robustness to traffic fluctuations. To perform run-time adaptation, ShaRE contains a scheduler Everest that monitors workload fluctuations and incorporates policies to decide what resources to allocate to which aggregates. The rest of the dissertation focuses on the design and implementation of various components of ShaRE.

2.3.1 Benefits of Run-time Adaptation

Although the capability of adapting processor allocations to aggregates at run-time appears to be an attractive feature, the actual benefits of such adaptation is a function of the total processor provisioning in the system and the robustness re-

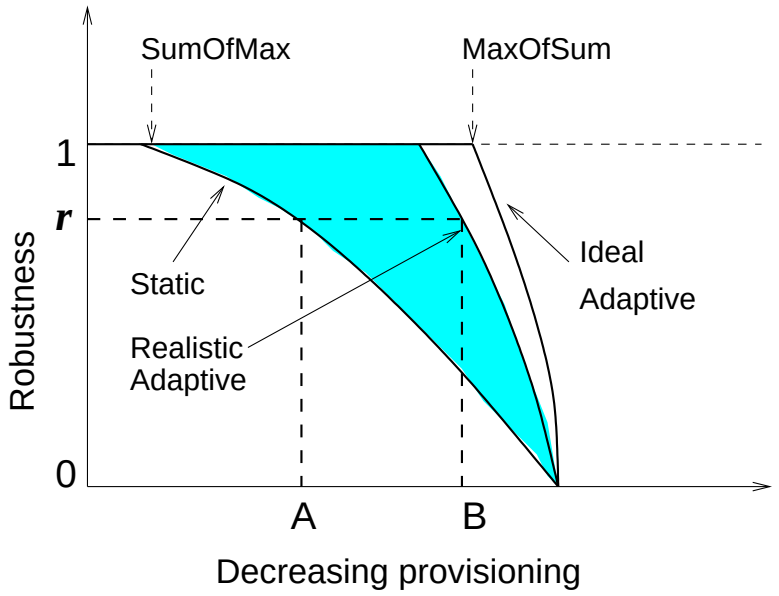


Figure 2.2: Benefits of Adaptation: Provisioning vs. Robustness.

quirement of the services hosted by the system. To understand the benefits better, we plot a hypothetical graph in Figure 2.2 that shows the relationship between system provisioning and the desired robustness for (1) a system that statically allocates processors to aggregates at the time of system initialization (or at service activation time), (2) an ideal adaptive system that adapts processor allocations at run-time, while incurring zero adaptation cost, in response to traffic fluctuations, and (3) a realistic adaptive system that incurs a cost every time it adapts processor allocations. Observe that the system provisioning decreases from left to right in the graph, and the robustness is a hypothetical metric between 0 and 1; different service specific metrics (such as fraction of packets processed within a delay tolerance, the deviation of average delay from a desired threshold, system throughput achieved as a fraction of the desired throughput, etc.) can be chosen to represent the desired robustness.

We define two specific points of interest on the provisioning axis—SumOfMax and MaxOfSum. SumOfMax represents the sum of processor provisioning required

by each aggregate to handle the maximum load observed by the aggregate at any instant of time. MaxOfSum represents the maximum provisioning required to handle the total load observed by all aggregates at each instant of time. Observe that SumOfMax is the minimum amount of provisioning needed by a static system to provide the maximum robustness, whereas an ideal adaptive system, by adapting processors at fine granularity, would require only MaxOfSum to provide such robustness.

The graph depicts that at high levels of provisioning (left of the graph), both Static and adaptive systems provide the maximum robustness. The robustness of a static system starts decreasing once the provisioning falls below SumOfMax. Whereas the robustness of an ideal adaptive system decreases only after the provisioning falls below MaxOfSum; between SumofMax and MaxOfSum, an adaptive system multiplexes processors to achieve robustness. A realistic adaptive system, however, pays a context switch overhead during processor adaptation, thereby requiring higher provisioning than ideal to guarantee maximum robustness. The area between the static and the realistic adaptive (the shaded region) is the benefit or opportunity of adaptation. The area between the lines for ideal adaptive and realistic adaptive represents the effect of adaptation costs; with increasing adaptation costs, the line for realistic adaptive shifts to the left, thereby decreasing the benefits of adaptation.

The metric on the x-axis (decreasing provisioning) can be replaced by "time before upgrade" of a virtualized router that has been installed in the field. With time (increasing from left to right in the graph), both the bandwidth demands and the service complexity increase, thereby making the effective provisioning of an installed system (without a hardware upgrade) decrease. Hence, even though a system is over-provisioned for the worst-case workload mix, with time, the system becomes less robust unless upgraded. Consider a horizontal line that represents a

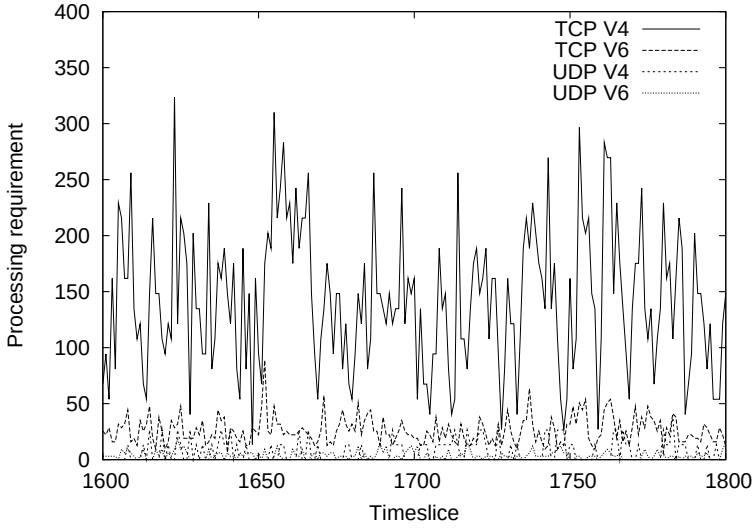


Figure 2.3: Variation in the processing requirements observed in 200 consecutive intervals (of length 1 ms) of a UNC trace [90] for four different aggregates of the IPv4-v6 gateway [98].

robustness of r. The graph shows that for guaranteeing the robustness level r, a static system has to be upgraded at time A. Whereas an adaptive system can wait till time B before upgrading the hardware resources.

We conclude that a system that adapts processor allocations to aggregates can be more robust to traffic fluctuations than a static system at any instant of time, and run-time adaptation can substantially increase the time to next upgrade of a given virtualized router.

2.3.2 Related Work

Traditionally, routers in the Internet have supported a single significant service (e.g., IP routing) [34, 63]. Cisco's IOS [34] and VERA [63] provision processors to meet the worst-case requirements of the significant service (e.g., IP routing), and host less frequently invoked services (e.g., routing table updates, router extensions like Network Address Translation, etc.) on a general-purpose CPU. Such a model is

inadequate for virtualized routers that support multiple significant services at high-throughput.

With the advent of programmable multi-processor router hardware that can support multiple network services simultaneously, several commercial and research efforts have begun developing tools to make such hardware easily programmable and achieve high throughput [5, 7, 8, 21, 24, 50, 70, 87, 91]. These environments provide toolkits for programming the router hardware, but they neither provide any support for adapting processor allocations at run-time nor provide any abstract interfaces that make the services portable across different instantiations (e.g. Intel's IXP2400 and a cluster of PCs) and versions of the multi-processor hardware (e.g. Intel's IXP2400 and IXP2800). For instance, NP-Click [87] and Intel's IXA-SDK [7] do not support run-time re-allocation of processors. NetBind [35] provides support for installing new services in a network-processor-based router during its operation, but no processor reallocation is performed in response to traffic fluctuations. Both NetBind [35] and Intel's IXA-SDK [7] provide no abstraction of hardware resources like processors, memory, and interconnects, thereby making programs non-portable.

Our work on building a run-time system for virtualized routers is motivated by two observations—(1) router hardware evolves with time; hence, providing abstract interfaces to make network services portable across different versions of the hardware is crucial, and (2) network traffic fluctuates significantly [68, 84, 111]. For instance, Figure 2.3 shows the fluctuation in processing requirements due to traffic variations for four different aggregates of an IPv4-v6 gateway service [98]. Hence, adapting processor allocations to services at run-time can reduce the provisioning level of a virtualized router and make the system robust to traffic fluctuations [68].

Chapter 3

The Everest Scheduler

In this chapter, we discuss in detail the problem of scheduling processor allocations to aggregates in multi-processor virtualized routers. We proceed in three steps. First, we develop a taxonomy that captures the behavior of multi-processor schedulers in four questions. The taxonomy enables us to position existing scheduling algorithms and identify their limitations. Second, guided by the taxonomy, we design Everest, an *agile* and *wary* scheduler suitable for processor management in virtualized routers built on today's state-of-the-art hardware. Everest's agility allows us to cope with stringent delay tolerance requirements, and difficult-to-predict and significant fluctuations in packet arrival rates at small timescales. Everest's wariness copes with the relatively large context switch overheads on today's state-of-the-art programmable router hardware. Finally, realizing that Everest is only a point solution suitable for today's virtualized routers, we perform a design space exploration of the behavior of the schedulers under various system, application and workload characteristics. We develop a parameterized scheduler framework that aids us in picking the more suitable scheduler behavior for given characteristics in today's, as well as future, routers. Such a framework copes with the evolving hardware architectures, changing application requirements, and diverse workload characteristics.

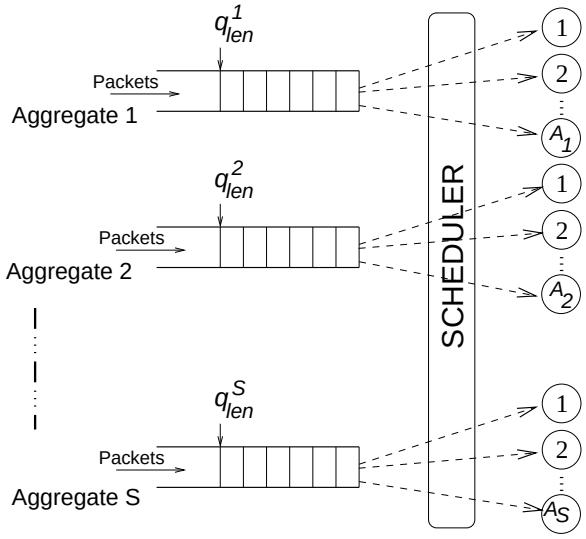


Figure 3.1: System model.

3.1 Requirements and System Model

Given a multi-processor virtualized router with N_p processors that hosts S_A aggregates (Figure 3.1), a scheduler that multiplexes the processors among the aggregates should meet two requirements.

1. Maximize the number of packets processed within a given *delay tolerance* to ensure that (1) the total delay perceived by packets that traverse multiple hops (routers) between two communicating end-points is within a range acceptable by the end-points, and (2) the queue (buffer) sizes maintained at the aggregates are small and do not overflow (thereby reducing the number of packets dropped).

2. Provide isolation of processor allocation to prevent damaging interactions and instabilities among aggregates in the presence of workload fluctuations and attacks; isolation ensures that the throughput guaranteed to an aggregate does not suffer when a different aggregate gets overloaded.

In order to develop a scheduler that meets the above requirements, we consider the following system model. Each processor has a local instruction cache to load the aggregates and all processors are identical in processing capacity and their ability to execute any aggregate[1]. At any instant, a subset of the processors in the system are allocated to each aggregate, and at most one aggregate can be loaded and run on a processor[2]. Switching a processor's context from one aggregate to another incurs an overhead of t_{sw} time units; i.e., the processor is unavailable to process packets for that duration. This overhead includes the time it takes to checkpoint and migrate the data of the old aggregate, and to load the code and data for the new aggregate in a processor's instruction and data caches. This hardware model closely approximates the IXP2400 multi-processor chip [57] that is representative of the evolving programmable multi-processor architectures for building high-performance routers.

A packet can be processed by one or more aggregates. Each aggregate is associated with a queue; packets are placed in the queue until a processor becomes available to process the packet. We assume that a packet queued at aggregate i can be served by any of the processors allocated to the aggregate i. Let the time taken to process a packet by aggregate i be t^i_{pkt} units. Packets at each aggregate are processed in the order of their arrival. After being processed by an aggregate, the packet is queued either for processing at another aggregate or for transmission at an outgoing link. We do not model the queue at the outgoing link and do not deal with packet scheduling. Several research efforts in the past have dealt with the packet scheduling problem at the outgoing link [32, 41, 44, 51, 55, 88, 93], and the

[1]The Everest scheduler we develop can be easily extended to processors with different processing capacities.

[2]Recall that the π-compiler makes the best utilization of pipelining and parallelism provided by the hardware while creating aggregates; hence it is best to load a single aggregate on each processor(Section 2.2). However, the Everest scheduler we develop in this chapter can be generalized to accommodate multiple aggregates per processor by appropriately scaling the throughput that each aggregate receives from the processor.

solutions can be applied in our context.

Let D^i denote the delay tolerance for aggregate i; this indicates that a packet should be processed by an aggregate within D^i time units of the packet's arrival[3] once it is enqueued for processing at aggregate i. Also, let R^i_{arr} (Δ) denote the rate of packet arrivals into the queue for aggregate i, estimated over Δ units of time.

In order to provide isolation of processor allocation among aggregates, let G^i denote the fraction of total processors in the system that are guaranteed[4] to be allocated to aggregate i even when some of the other aggregates are overloaded $(\sum_i G^i \leq 1)$. Let N^i_{reqd} denote the processor requirement of aggregate i to process all packets queued at aggregate i within their delay tolerance D^i. We define N^i_{min} as the minimum of the current processor requirement and the guaranteed allocation for aggregate i.

$$N^i_{min} \quad = \quad min\ (N^i_{reqd},\ G^i \times N_p) \tag{3.1}$$

At any instant, let N^i_{alloc} denote the number of processors allocated to an aggregate that can either be actively processing packets or can be switching their context to aggregate i. Then, to provide isolation among aggregates, the following condition on N^i_{alloc} must hold.

$$N^i_{min} \ \leq \ N^i_{alloc} \ \leq \ N_p \tag{3.2}$$

Table 3.1 summarizes the different system parameters.

[3]The end-to-end delay perceived by a packet is the sum of delays perceived at each aggregate, and the delay incurred at the incoming and outgoing links. The problem of distributing a given end-to-end delay tolerance at the router across different aggregates is an orthogonal problem, and similar problems have been addressed in the past [101]. We do not focus on the problem in this dissertation; we assume that each aggregate i attempts to provide a delay tolerance D^i for its packets.

[4]The guaranteed allocation for different aggregates can be derived from minimum throughput requirements requested by a service-provider for the service. For example, an overlay service may request a virtualized router to provide a minimum guaranteed bandwidth of 1 Gbps.

Parameter	Description
N_p	Number of processors in the system
S_A	Number of aggregates
R^i_{arr} (Δ)	Estimated packet arrival rate for aggregate i measured over an interval Δ
t^i_{pkt}	Processing time per packet for aggregate i
t_{sw}	Context switch overhead
D^i	Delay bound for aggregate i
G^i	Guaranteed allocation—fraction of processors that are allocated to aggregate i when all aggregates are overloaded
N^i_{reqd}	Processors required by i at any instant
N^i_{alloc}	Processors allocated to i at any instant

Table 3.1: System model parameters for Everest design.

3.2 Design Considerations

Our objective is to design a scheduler that meets the above two requirements. Although several past schedulers in various domains—real-time systems [27, 52, 74, 94], conventional operating systems [37, 38, 103], web servers [23, 29, 39, 40, 62, 100, 106], software-based routers [85, 92], storage systems [60, 76], and multi-media servers [54, 107]—have been proposed and applied to multiplex processing resources among competing services, virtualized routers introduce several unique characteristics that render past solutions ineffective in meeting the two requirements. In the rest of this section, we discuss the unique characteristics of virtualized routers, identify the properties that a scheduler for such routers should exhibit, and discuss why past scheduling solutions are inadequate.

3.2.1 Unique characteristics

- The arrival rate of packets at virtualized routers is at least two to three orders of magnitude greater than the arrival rate of requests for other hosting environments (e.g., millions of packets per second vs. thousands of requests

per second in a web hosting environment). In such scenarios, often, the inter-arrival time of packets is much smaller than the processing time of a packet ($t_{iat} \ll t_{pkt}$). Further, network traffic is bursty and hard-to-predict, especially at fine timescales (upto 100s of milliseconds) [84, 111].

- The context-switch overhead of reallocating processing resources from one aggregate to another in current state-of-the-art hardware [57] is large and is often two to three orders of magnitude greater than the time required to process a packet; on an Intel®'s IXP2400 network processor, loading the instruction store of a *micro-engine* with the code for a new aggregate incurs a context-switch time that is often greater than $1ms$ (see Chapter 4), while a packet can often be processed within a few microseconds ($t_{sw} \gg t_{pkt}$).

- Network services are required to process packets within a small delay from their arrivals. In contrast to conventional operating systems and hosting environments where a relatively large delay (e.g., $10ms$) is tolerable even for interactive applications, in this environment adding even a $1ms$ scheduling delay per hop (router) for each packet is prohibitive. The delay tolerance D is of the same order as t_{sw} ($D \sim t_{sw}$).

3.2.2 Desirable Properties of a Scheduler

To address the two requirements in virtualized routers, we identify two desirable properties that a scheduler should exhibit.

1. **Agility:** To meet the two requirements, a scheduler should determine the processor allocation at the right instant and allocate/release the right number of processors. Since packets can arrive at high rates ($t_{iat} \ll t_{sw}$), and $D \sim t_{sw}$, it is crucial to detect as early as possible if new processor allocations are required to minimize the number of packets whose delay will exceed D.

2. **Wariness:** Since $t_{pkt} \ll t_{sw}$, and each context switch makes a processor unavailable for t_{sw} time units, context switches reduce a processor's effective capacity available for servicing packets. In such a scenario, allocating/releasing processors too frequently can reduce the total effective processing capacity of the virtualized router; it is crucial to be cautious and switch a processor's context only when necessary.

Observe that, since context switch overhead is large, and the arrival rate of traffic is high and hard to predict, isolation can be provided trivially by always allocating processors in proportion to guaranteed allocation; however, doing so will not maximize the number of packets that are processed within their delay tolerance.

3.3 Multi-processor Scheduler Taxonomy

To understand the drawbacks of past scheduling solutions in satisfying the two requirements, and guide the design of a scheduler that exhibits the two desirable properties—agility and wariness—we first develop a taxonomy of multi-processor schedulers. Given a set of processors to be multiplexed among a set of aggregates, any multi-processor scheduler should address four questions.

1. When does an aggregate become eligible to receive additional processors?

   To address this question, the scheduler *monitors* the system, and based on certain metric, decides *when to trigger* additional allocations.

2. When does an aggregate release a processor?

   In addressing this question, the scheduler can either let the aggregate release a processor *voluntarily* or *force* (interrupt) the aggregate to release it.

3. When an aggregate becomes eligible, how is a victim aggregate (from whom a processor is reclaimed) chosen?

A scheduler should address this question under two scenarios— (1) when the system contains free processors (i.e., the system is *underloaded*), and (2) none of the processors are free (i.e., the system is *overloaded*).

4. When an aggregate releases processors, how is a beneficiary aggregate (who receives the processors) chosen?

   This question needs to be addressed under two scenarios— (1) No aggregates are currently eligible to receive additional processors (*No-demand*), and (2) some aggregates are eligible to receive additional processors (*In-demand*).

Observe that while the first two questions deal with per-aggregate decisions of the scheduler, the last two questions address how processors should be multiplexed across multiple aggregates.

This taxonomy allows us to systematically position the behavior of several known schedulers. While there have been numerous schedulers proposed in the past, they can be represented as variants of one of the following classes. Table 3.2 provides a summary of these schedulers.

Static allocation State-of-the-art programming environments for multi-processor routers perform the allocation of processors to aggregates statically (at design-time) [24, 87, 91]. During the system operation, no aggregate becomes eligible to receive any additional processors, and no aggregate releases processors either voluntarily or is forced by an external event. Hence, no victim/beneficiary selection is necessary.

Quantum-based schedulers Quantum-based schedulers have been widely used in conventional operating systems for several years [37, 38, 43, 54, 60, 85, 92, 103, 107]. In quantum-based scheduling, any aggregate with a non-zero queue is eligible to receive an additional processor. Once allocated, an aggregate releases a proces-

sor voluntarily when its queue becomes empty (i.e. no more work left to do for the current aggregate), or is forced to release a processor when the processor has serviced packets for a quantum (Q) amount of time. Once a processor becomes free, it is allocated to one of the eligible aggregates; the beneficiary is chosen using one of several policies such as round-robin, proportionate share. If no aggregate is eligible (i.e. there is no work to do for any aggregate), the processor remains with the current aggregate till new work arrives; this choice improves the chance of maximizing the instruction and data cache locality. By allowing a processor to process a number of packets or requests (or for a time quantum Q, $Q \gg t_{sw}$) of an aggregate before switching it to a different aggregate, Quantum schedulers amortize the context switch overhead.

Cohort scheduler Cohort scheduling [71], proposed for servers, maximizes the data and instruction cache locality by batching the execution of similar operations while processing different server requests. Since the overhead of switching a processor from one aggregate's context to another incurs significant overhead in a virtualized router, applying cohort scheduling promises to minimize the number of context switches. While the processor allocation eligibility and beneficiary selection criteria are similar to quantum schedulers, a cohort scheduler only allows voluntary release of processors when the queue for the aggregate becomes empty. And since processors can not be released by force, victim selection is not a part of the Cohort scheduler.

Observation-based schedulers Observation-based schedulers [23, 29, 39, 40, 59, 62, 100, 106] utilize long-term weighted average of observed workload and performance (e.g., arrival rate, average or worst-case latency in serving a request) for past requests as a metric to predict the workload requirements and to allocate processors for the future. For instance, Chandra et al [39] and Abdelzaher et al [23] measure the average latency and arrival rate in fixed intervals, and when the latency exceeds a

threshold, vary the proportional allocation of resources across services. SEDA [106] monitors the 90-th percentile latency of a number of requests and allocates extra resources if the latency exceeds a threshold. In all the different observation-based schedulers, aggregates become eligible when the chosen metric exceeds a threshold. In the rest of the chapter, we choose SEDA as a representative of observation-based schedulers. In SEDA, aggregates release processors to a free pool when they are idle for a certain period of time. SEDA does not explicitly specify if processors/threads can be reclaimed by force, and hence also does not specify victim selection procedure; we extend SEDA to include victim selection later in this chapter.

3.3.1 Limitations of Past Schedulers

Static Allocation Static allocation, by definition, does not provide the desired agility to minimize the packets that exceed their delay tolerance. Hence, one way to guarantee robustness to fluctuations in the arrival rate for different aggregates is to provision sufficient number of processors to handle the expected *maximum* load for each aggregate. In virtualized routers, since new services can be deployed at any time during the system operation, and old services can be removed at any time, determining static allocation is inherently not possible. Even when all the services are known a priori, allocating statically for the worst case requirement of each aggregate within a service causes significant wastage of processing resources than an adaptive system that reaps the benefits of statistical multiplexing [68]. Alternately, for any given level of processor provisioning, a static system can be substantially less robust to traffic fluctuations than an adaptive system(See Section 2.3.1).

Quantum-based Schedulers In a virtualized router, where the delay tolerance D is of similar magnitude as the time t_{sw} to switch the context of a processor from one aggregate to another, the choice of the quantum size is crucial to maximize the number of packets that are processed within an aggregate's delay tolerance.

Choosing smaller quantum size makes the scheduler more agile but less wary; it results in significant reduction in processing capacity due to context switching. For instance, request-level schedulers [38, 107] may switch processors on every request; since in virtualized routers, the time t_{pkt} to service a packet is often significantly smaller than t_{sw}, request-level schedulers can waste significant processing capacity in context switching. Choosing a large quantum size ($Q \gg t_{sw}$) makes the scheduler more wary; it amortizes the context switch overhead over several requests. However, the scheduler becomes less agile, thereby letting packets miss their delay tolerance; packet delays can be of the order of Q. Further, since network traffic fluctuates, any single quantum size may not be the best value at all times.

Cohort Scheduler Cohort scheduling has two drawbacks. First, since Cohort scheduling does not support victim selection, the time between when an aggregate becomes eligible to receive a processor and when a processor is actually allocated to the aggregate is not bounded; hence, Cohort scheduling by definition does not exhibit the agility property. As a result, a large number of packets can observe delays larger than D. Second, a single aggregate can get overloaded such that all processors get allocated to it and are never released for other aggregates; if processors can not be taken away by force, the scheduler cannot provide isolation across aggregates.

Observation-based Schedulers In virtualized routers, because $t_{iat} \ll t_{sw}$, waiting until packets observe degraded performance makes observation-based schedulers less agile than necessary; many packets that arrive after the packet that triggered processor allocation can experience delay greater than D^i before the new processor is ready to process packets. Delayed allocation can result in queue build-up and consequently more packets can exceed their tolerance. Whereas, reacting eagerly based on past measurements can be inaccurate because of fluctuating and hard-to-predict packet arrival rate [84, 111], thereby leading to unnecessary context switches.

	Allocation Eligibility		Processor Release		Victim		Beneficiary	
Algorithm ↓	monitor	trigger	voluntary	forced	under-load	overload	No-demand	In-demand
Static	–	$q_{len} = \infty$	–	–	–	–	–	–
Quantum	q_{len}	$q_{len} > 0$	$q_{len} = 0$	$T_{alloc} > Q$	–	Prop share	Locality	Prop share
Cohort	q_{len}	$q_{len} > 0$	$q_{len} = 0$	–	–	–	Locality	RR
Observation (e.g. SEDA)	pkt delay	90-th pkt delay $> \delta$ and $T_{lastalloc} > t_{sw}$	proc idle $> \epsilon$	–	–	–	Free pool	Prop share

Table 3.2: Taxonomy of conventional multi-processor schedulers.

3.4 Design of Everest

In this section, we present Everest, a multi-processor scheduler that exhibits both agility and wariness. Algorithm 1 shows the pseudo-code for the basic behavior of Everest. Everest addresses allocation eligibility in lines 3 and 4, voluntary processor release in lines 6 and 7, victim selection in line 5, and beneficiary selection in line 8. Note that forced processor release is a part of victim selection. In what follows, we derive the details of allocation eligibility, processor release, beneficiary and victim selection. Since the derivation of allocation eligibility and processor release are per-aggregate, for brevity we will eliminate any reference to a specific aggregate (and drop the superscript i) in the next two subsections. We will re-introduce i for beneficiary/victim selection. Also, to represent processors allocated to an aggregate, we use the variable j interchangeably with N^i_{alloc} for brevity.

Algorithm 1 Everest Algorithm

```
1:  while (1) do
2:    for (each aggregate i) do
3:      if (q_len > Q^j_lim) then
4:        Make aggregate i eligible to receive processors
5:        victim_select(i)
6:      else if (q_len = 0) then
7:        Mark or unmark processors such that at most k^j are marked
8:        N^i_marked ← k^j
9:        beneficiary_select(i)
10:     end if
11:   end for
12: end while
```

3.4.1 Allocation Eligibility

Since $t_{iat} \ll t_{pkt}$, waiting until a packet observes a delay greater than D can cause significant number of packets to exceed their delay tolerance. At the same time, it is hard to predict the future arrival rate of traffic at fine timescales. Hence,

instead of determining the need for allocating additional processors based on past observations (perceived delay or arrival rate in previous intervals), Everest achieves the desired agility and wariness by monitoring and reacting to the *current* state of the system. In particular, given a delay tolerance D on an aggregate and current level of processor allocation j, Everest determines a threshold Q^j_{lim} on each aggregate's queue. Everest monitors the length of the packet queue for each aggregate and declares an aggregate eligible to receive a new processor *if and only if* the aggregate's queue length exceeds the threshold Q^j_{lim}.

Deriving Q^j_{lim} The queuing delay observed by a packet depends on the queue length when the packet arrives and the rate at which packets are serviced from the queue. The rate at which packets are serviced from the queue is a function of the number of processors j allocated to the aggregate. In particular, since each processor can service a packet in t_{pkt} time[5], the number of packets that can be serviced within any x units of time is

$$P^j_{dep}(x) = \left\lfloor \frac{x}{t_{pkt}} \right\rfloor \times j \tag{3.3}$$

We define Q^j_{lim} to be the maximum queue length that j processors can process within the delay tolerance D. If there are Q^j_{lim} packets in the queue and the aggregate is allocated j processors (that are currently processing j packets), then the total number of packets in the system for the aggregate is $Q^j_{lim} + j$. Hence, the maximum length Q^j_{lim} that a queue can grow to without exceeding the delay tolerance for any packet is given by:

$$\begin{aligned} Q^j_{lim} + j &= P^j_{dep}(D) \\ \Rightarrow Q^j_{lim} &= j * \left(\left\lfloor \frac{D}{t_{pkt}} \right\rfloor - 1 \right) \end{aligned} \tag{3.4}$$

[5]We assume that all processors are identical in processing capacity. The derivation can be easily extended to heterogeneous processors.

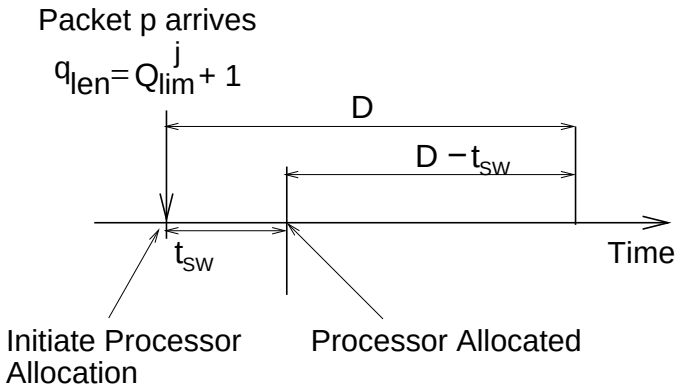


Figure 3.2: Processor allocation timing diagram.

Observation Suppose the arrival of a packet p causes the queue length to become $Q_{lim}^{j} + 1$ (and hence the total number of packets in the aggregate to become $Q_{lim}^{j} + 1 + j$). Then, with only j allocated processors, the delay incurred by packet p would exceed D. To ensure that packet p can be serviced prior to its delay bound, an additional processor must be allocated. Observe, however, that since allocating a processor incurs an overhead of t_{sw}, packets will continue to be processed with j processors for t_{sw} duration; only after that time, $(j+1)$ processors will begin processing packets(see Figure 3.2). Hence, packet p will be processed before delay D only if the total packets processed by j processors in D units of time and the $(j+1)^{th}$ processor in $D - t_{sw}$ units of time include the packet p, i.e.,

$$\begin{aligned} Q_{lim}^{j} + 1 + j &\leq P_{dep}^{j}(D) + P_{dep}^{1}(D - t_{sw}) \\ \Rightarrow Q_{lim}^{j} + 1 + j &\leq j * \left\lfloor \frac{D}{t_{pkt}} \right\rfloor + \left\lfloor \frac{D - t_{sw}}{t_{pkt}} \right\rfloor \end{aligned} \tag{3.5}$$

By substituting Q_{lim}^{j} from Equation 3.4 and simplifying, the above requirement reduces to

$$D \geq t_{pkt} + t_{sw} \tag{3.6}$$

Equation 3.6 indicates that if $D \geq (t_{pkt} + t_{sw})$, then for all values of j the packet that triggers adaptation will be serviced before its delay tolerance.

When $D < (t_{pkt} + t_{sw})$, the packet that triggered processor allocation as well as some of the follow-on packets may experience delays greater than the tolerance. In this case, it is possible to adjust the queue threshold to allocate processors proactively (in anticipation of the event that queue length will exceed Q^j_{lim}) to ensure that no packets experience delays greater than the tolerance; however, doing so requires prediction of the future packet arrival rate (generally based on past measurements). Unfortunately, because of the significant fluctuations in packet arrival rates at fine timescales, such predictions are often inaccurate and lead to unnecessary processor allocations. To minimize such allocations, Everest allocates a processor *only* upon receiving a packet that results in the queue length exceeding the threshold Q^j_{lim}. We design a proactive version of Everest in Appendix 1 and demonstrate that being proactive only increases the number of packets that exceed their delay tolerance.

3.4.2 Processor Release

Voluntary release Everest marks a processor allocated to aggregate i for release *only* after the queue for aggregate i becomes empty. A queue becomes empty when the processing capacity allocated to an aggregate exceeds the arrival rate of packets for the aggregate. Further, Everest marks for release only those processors that are not required to process packets at the current arrival rate. In particular, if $R_{arr}(\Delta)$ is the current arrival rate of packets for aggregate i estimated over an interval Δ , then Everest marks k^j processors for release only if:

$$\begin{aligned} P^{(j-k^j)}_{dep}(t_{pkt}) \;&\geq\; R_{arr}(\Delta) \times t_{pkt} \\ \Rightarrow k^j \;&\leq\; j - R_{arr}(\Delta) \times t_{pkt} \end{aligned} \tag{3.7}$$

The processors marked for release are retained with aggregate i until some other aggregate becomes eligible to receive processors. If the same aggregate becomes eligible to receive processors before the marked processors are context-switched to

another aggregate, then the necessary number of processors are unmarked without incurring a context switch overhead (exploiting locality); line 7 of Algorithm 1 represents this behavior. These design choices make Everest wary; they reduce the number of context-switches and also make Everest non-work-conserving. Since the processors are only marked for release but not switched immediately to another aggregate (unless an eligible aggregate exists), Everest can better tolerate inaccuracies in $R_{arr}(\Delta)$ estimates.

Forced release If an aggregate is allocated processors beyond its guaranteed allocation, Everest can reclaim the extra processors (during victim selection) through an interrupt to satisfy the guaranteed allocation of a different aggregate; this enables Everest to achieve the desired isolation across aggregates.

3.4.3 Victim Selection

When an aggregate requires an additional processor, a victim aggregate is selected to reclaim a processor. At this instant, a system can be either underloaded or overloaded; the system is underloaded if there are processors marked for release, otherwise it is overloaded. A victim is selected in each of the states as follows.

Underload When the system is underloaded, our objective in selecting a victim aggregate is to pick the processor that has the least *utility* at that instant for context-switching; we select the aggregate that has processors marked for release and whose last time of processor allocation is the oldest (LRA—Least Recently Allocated, like LRU). To do so, we maintain a variable $T^i_{lastalloc}$ that stores the timestamp of the last processor allocation. The aggregate with marked processors and smallest $T^i_{lastalloc}$ is chosen as the victim. Using LRA reduces the chances of context switching due to transient fluctuations in traffic. Also, this policy ensures that processors remain with an aggregate unless the aggregate is chosen as a victim, thereby maximizing locality;

only a subset of the processors that need to be re-allocated between aggregates pay the context switch overhead.

Overload To handle overloaded condition, Everest uses the following two-step policy.

1. Everest ensures that each aggregate is allocated processors equal to the minimum of the guaranteed allocation to the aggregate and its current requirements.

$$N^i_{newalloc} \quad = \quad min\ (N^i_{reqd},\ G^i \times N_p)$$

2. Let $N_{extra} = N_p - \sum_i N^i_{newalloc}$, denote the extra processors. Our objective in distributing the extra processors is to minimize the maximum queue build-up for each aggregate to ensure that minimum packets (1) experience delays greater than their delay tolerance, and (2) get dropped due to queue overflow. To do so, we allocate each free processor to the aggregate that has the maximum *residual queue length*—the difference between the queue length and the current threshold on the aggregate. This policy has a bias towards aggregates with longer backlogged queues. When there are two aggregates with maximum residual queue length, the aggregate with $N^i_{alloc} > N^i_{newalloc}$ is chosen to avoid a context switch. If neither aggregate satisfies the condition, an aggregate is chosen arbitrarily. Observe that by choosing either aggregate arbitrarily, the extra packets that meet their delay tolerance due to the new allocation is the same.

Algorithm 2 shows the pseudo-code for victim selection. Observe that when all aggregates are overloaded and $G^i = 1$, only step 1 gets executed; in this case, any adaptive system can do only as good as a system that statically allocates processors in the ratio of G^i. When a subset of the aggregates are overloaded, the first step

ensures that all aggregates meet their requirements at least up to the guaranteed allocation, thereby providing the desired isolation.

Algorithm 2 *victim_select*(i)

```
1: if (marked processors exist) then
2:    /* Underload condition */
3:    pick aggregate k with N^k_marked > 0 and smallest T^k_lastalloc
4:    N^i_alloc ← N^i_alloc + 1
5:    N^k_alloc ← N^k_alloc − 1
6:    N^k_marked ← N^k_marked − 1
7:    context switch a processor from k to i
8:    set T^i_lastalloc to current time
9:    if (N^i_alloc ≥ N^i_reqd) then
10:      Make aggregate i ineligible to receive processors
11:   end if
12: else
13:   handle_overload()
14: end if
```

3.4.4 Beneficiary Selection

When an aggregate i releases processors, beneficiary aggregates are chosen to receive the released processors. At this instant, the system can be in two states, (1) there is no eligible aggregate (No-demand) to receive processors, and (2) there are eligible aggregates (In-demand) to receive the processors.

No-demand If there are no eligible aggregates, processors remain allocated to the current aggregate i (and process any future packets) until an aggregate k becomes eligible and selects i as the victim aggregate. This design decision avoids unnecessary context switches.

In-demand Two scenarios arise in this case. First, if the requirements of the eligible aggregates are less than or equal to the marked processors N^i_{marked}, then all the requirements are satisfied by a subset of aggregate i's marked processors. The

Algorithm 3 *handle_overload*()

1: **for** each aggregate (k) **do**
2: $N^k_{newalloc} \leftarrow N^k_{min}$
3: **end for**
4: $N_{extra} = N_p - \sum_k N^k_{newalloc}$
5: **while** ($N_{extra} > 0$) **do**
6: pick aggregate k with maximum $q^k_{len} - Q^{N^k_{newalloc}}_{lim}$
7: $N^k_{newalloc} \leftarrow N^k_{newalloc} + 1$
8: $N_{extra} \leftarrow N_{extra} - 1$
9: **end while**
10:
11: /* $N_{newalloc}$ has the new allocation */
12: /* N_{alloc} has the current allocation */
13: **for** (each aggregate k with $N^k_{newalloc} > N^k_{alloc}$) **do**
14: pick aggregate m with $N^m_{newalloc} < N^m_{alloc}$
15: $N^k_{alloc} \leftarrow N^k_{alloc} + 1$
16: $N^m_{alloc} \leftarrow N^m_{alloc} - 1$
17: context switch a processor from m to k
18: set $T^k_{lastalloc}$ to current time
19: **end for**
20:
21: **for** (each aggregate k) **do**
22: **if** ($N^k_{alloc} \geq N^k_{reqd}$) **then**
23: Make aggregate k ineligible to receive processors
24: **end if**
25: **end for**

rest of the marked processors (if any) will remain allocated to aggregate i. Second, if the requirements of the eligible aggregates are higher than the marked processors, Everest uses the same two-step (overload) policy as in victim selection. This policy ensures that isolation is maintained during overload, and the number of packets that exceed their delay tolerance is minimized. Algorithm 4 shows the pseudo-code for beneficiary selection.

Algorithm 4 $beneficiary\_select(i)$

```
1: if (No-demand) then
2:    return
3: else if (demand of eligible aggregates ≤ N^i_marked) then
4:    /* N^i_marked is number of marked processors for release */
5:    for (each eligible aggregate k) do
6:       N^k_alloc ← N^k_alloc + 1
7:       N^i_alloc ← N^i_alloc − 1
8:       N^i_marked ← N^i_marked − 1
9:       context switch a processor from i to k
10:      set T^k_lastalloc to current time
11:      if (N^k_alloc ≥ N^k_reqd) then
12:         Make aggregate k ineligible to receive processors
13:      end if
14:   end for
15: else
16:   /* Demand > marked processors */
17:   handle_overload()
18:   N^i_marked ← 0
19: end if
```

	Allocation Eligibility		Processor Release		Victim		Beneficiary	
Algorithm ↓	monitor	trigger	voluntary	forced	under-load	overload	No-demand	Demand
Static	–	$q_{len} = \infty$	–	–	–	–	–	–
Quantum	q_{len}	$q_{len} > 0$	$q_{len} = 0$	$T_{alloc} > Q$	–	Prop share	Locality	Prop share
Cohort	q_{len}	$q_{len} > 0$	$q_{len} = 0$	–	–	–	Locality	RR
Observation (e.g. SEDA)	pkt delay	90-th pkt delay $> \delta$ and $T_{lastalloc} > t_{sw}$	proc idle $> \epsilon$	–	–	–	Free pool	Prop share
Everest	q_{len}	$q_{len} > Q^j_{lim}$	Mark k^j when $q_{len} = 0$ and $R_{dep} > R_{arr}$	Interrupt	LRA	Prop share	Locality	Prop share
Everest-LD	pkt delay	pkt delay $> \delta$ and $T_{lastalloc} > t_{sw}$	Mark k^j when $q_{len} = 0$ and $R_{dep} > R_{arr}$	Interrupt	LRA	Prop share	Locality	Prop share
Cohort-E	q_{len}	$q_{len} > 0$	$q_{len} = 0$	–	–	–	Locality	Prop share
Everest-AE	q_{len}	Hi: $q_{len} > Q^j_{lim}$ Lo: $q_{len} > 0$	$q_{len} = 0$	Interrupt	LRA	Prop share	Locality	Prop share

Table 3.3: Taxonomy of multi-processor schedulers for comparison.

3.5 Discussion

In this section, we provide a qualitative comparison of Everest with past schedulers (Table 3.3). Recall that Quantum and Cohort schedulers make aggregates eligible eagerly when $q_{len} > 0$. Both schedulers release processors as soon as an aggregate's queue becomes empty. This eager behavior can lead to substantially larger number of context switches than necessary to process packets within their delay tolerance. Since a processor becomes unavailable for t_{sw} time units during context switch, eagerly switching processors can reduce the processing capacity of the system, thereby increasing the number of packets that experience delay greater than their tolerance. In contrast, Everest makes an aggregate eligible *only* when it receives packets that will experience delay greater than their tolerance, and releases processors only after the requirement drops below the processing capacity for some amount of time; such lazy release of processors absorbs small traffic fluctuations and thereby reduces context switches.

Changing the processor allocation eligibility of Everest to monitor and react to packet delays allows us to model an observation-based scheduler like SEDA [106]. Everest-LD is similar to Everest, but with two crucial differences. First, Everest-LD adapts processor allocations only *after* a packet suffers delay greater than its tolerance. Second, in order to maintain stability of processor allocation, Everest-LD separates two consecutive processor allocations to an aggregate by at least t_{sw} units of time to ensure that the previous allocation has taken effect prior to initiating a new allocation. This is different from Everest where a processor allocation is triggered as soon as the queue length exceeds Q^j_{lim}, independent of when the previous allocation was done. As a result, Everest is more agile than Everest-LD. Also, SEDA and Everest-LD differ in the release policy. First, while SEDA moves released processors into a free pool, Everest-LD keeps the processors allocated to the current aggregate unless another aggregate is eligible. Second, SEDA [106] does not ex-

plicitly mention any provision for forced release of threads once allocated, whereas, Everest-LD can reclaim a processor from an aggregate through an interrupt.

The taxonomy allowed us to identify the dimensions in which the Cohort scheduler is deficient for providing isolation; we enhanced the scheduler (to Cohort-E) to proportionally allocate processors at each adaptation step; a released processor is allocated to an aggregate such that at that instant the processors are allocated in proportion to the queue lengths (Note the difference between Cohort and Cohort-E in Table 3.3). However, since $q_{len} > 0$ is the only eligibility criteria, Cohort-E cannot determine when to interrupt processors of an aggregate that is over-allocated in order to process packets of a different aggregate (that is under-allocated) within the delay tolerance. Since no processors can be interrupted, no victim selection is specified. As a result, even Cohort-E can not provide the desired isolation. For instance, a single overloaded aggregate can get all processors allocated and never release them for other aggregates.

We define a work-conserving variant of Everest, Everest-AE, that fixes the drawback of Cohort-E in providing isolation. The allocation eligibility of Everest-AE is a prioritized combination of Everest and Cohort-E. In particular, aggregates with queue lengths exceeding the threshold Q^j_{lim} are designated as eligible to receive another processor with a higher priority, and all aggregates with a non-zero queue length are eligible with a lower priority. Processors are released by an aggregate as soon as its queue becomes empty without waiting for the t_{sw} time, just like Quantum or Cohort. The enhanced eligibility criteria allows Everest-AE to interrupt processors in order to provide isolation.

Everest-AE's eligibility and release criteria, however, make it more aggressive than Everest. When the context switch overhead is large, being eager like Everest-AE can reduce the effective processing capacity for servicing packets, thereby increasing the number of packets that exceed their delay tolerance.

3.6 Evaluation

In this section, we demonstrate the effectiveness of Everest in multiplexing processors among competing aggregates using trace-driven simulations.

3.6.1 Experimental Methodology

Aggregate	t_{pkt} (μs)	Aggregate	t_{pkt} (μs)
v4→v6/TCP	18.22	v6→v4/TCP	10.09
v4→v6//UDP	13.27	v6→v4/UDP	9.72
v4→v6//ICMP	4.64	v6→v4/ICMP	1.78

Table 3.4: t_{pkt} for IPv4-v6 gateway aggregates.

Network service: We present our results in the context of a canonical network service—an IPv4-v6 gateway (NAT-PT) [98] that enables seamless communication across devices that only support IPv4 with devices that only support IPv6 through protocol translation. Further, depending on whether the packet type is TCP, UDP or ICMP, the service translates transport identifiers (e.g. TCP and UDP port numbers, ICMP query identifiers), and updates the header checksums [98]. We profile a NAT-PT service [10] and measure the computation time required to process a packet at each of the six aggregates (See Table 3.4). Observe that virtualized routers can host multiple such services, each containing several aggregates. Greater number of co-hosted services increases the benefits of multiplexing.

Traces: We evaluate Everest using traffic traces collected from an edge link connecting the University of North Carolina at Chapel Hill (UNC) to its network service provider [90]. In addition to using these traces as is, we also derive representative traces for higher bandwidth links (e.g., an OC-48 link) using well-known trace scaling techniques [30]; we scale the number of flows observed in unit time, and retain the flow level properties. Table 3.5 shows the characteristics of various traces we

Trace Name	Peak b/w (in Gbps)	Duration (seconds)	#pkts (million)	Collected on
UNC-2.5	2.5	42.4	30	29th Sep 2000, 11am
UNC-1	1	83.2	30	29th Sep 2000, 11am
UNC-10	10	21.1	60	29th Sep 2000, 11am
UNC-58	2.5	70.1	30	25th Sep 2000, 8.30am
UNC-51	2.5	59.3	30	25th Sep 2000, 11am

Table 3.5: Trace characteristics.

use. Each UNC trace consists of two packet sequences—incoming and outgoing. To emulate the behavior of a typical IPv4/v6 gateway, we treat all packets received on incoming link as IPv4 packets, and all packets on the outgoing link as IPv6 packets.

Proportional share: Through trace analysis, we determine the maximum processor requirement N^i_{max} of each aggregate to process all the packets within their delay tolerance. The proportional share for each aggregate i is defined as

$$G^i = \frac{N^i_{max}}{\sum_i^S N^i_{max}} \tag{3.8}$$

3.6.2 Benefits of Everest

The experiments in this section demonstrate the benefits of the two properties on which the design of Everest is based. Figure 3.3 compares the percentage of packets that suffer delays greater than the aggregate's tolerance with different schedulers in a system with 16 processors, $t_{sw} = 2ms$, and UNC-2.5 trace[6]. For Everest and Everest-LD, the arrival rate $R_{arr}(\Delta)$ for each aggregate is estimated as the rate of packets received in the last t_{sw} amount of time, i.e., $\Delta = t_{sw}$. For Quantum schedulers, we experiment with different quantum sizes. For brevity, we only show

[6]The parameter setting $t_{sw} = 2ms$ represents the time it takes to switch the context of a processing core from one aggregate to another on the Intel's IXP2400 platform [57]. We perform a sensitivity analysis with different values of t_{sw} in the next section.

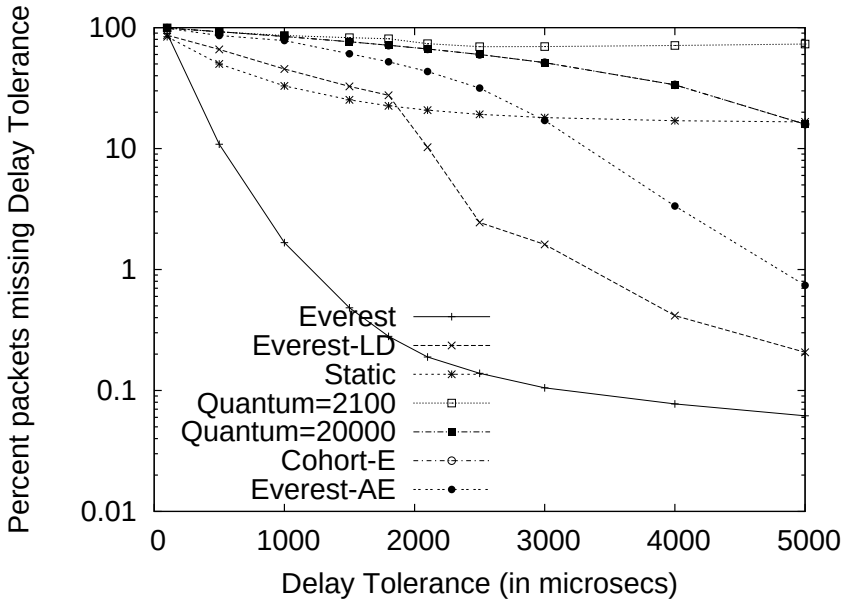


Figure 3.3: Effectiveness of Everest in reducing the number of packets that miss their delay bounds with increasing delay tolerance.

results for two quantum sizes; the results for other quantum sizes are similar.

The graph shows that Everest reduces the number of packets that suffer delays greater than D by orders of magnitude compared to other schedulers. The difference between Everest and Everest-LD demonstrates the benefit of *agility*. To explain the difference better, Figure 3.4 plots the distribution of queue lengths for various schedulers for the entire trace duration. Since processors are allocated late with Everest-LD, queues at aggregates build up and cause greater number of packets to exceed their delay tolerance. Increased number of packets missing delay tolerance also leads to triggering of greater number of processor context switches. This phenomenon can be observed in Figure 3.5, that plots the total number of context switches incurred using different algorithms, with varying delay tolerance.

The difference between the work-conserving schedulers (Quantum, Cohort-E and Everest-AE) and Everest in Figure 3.3 demonstrates the benefits of *wariness*. Quantum-based schedulers (with quantum sizes of $2.1ms$ and $20ms$), the Cohort-

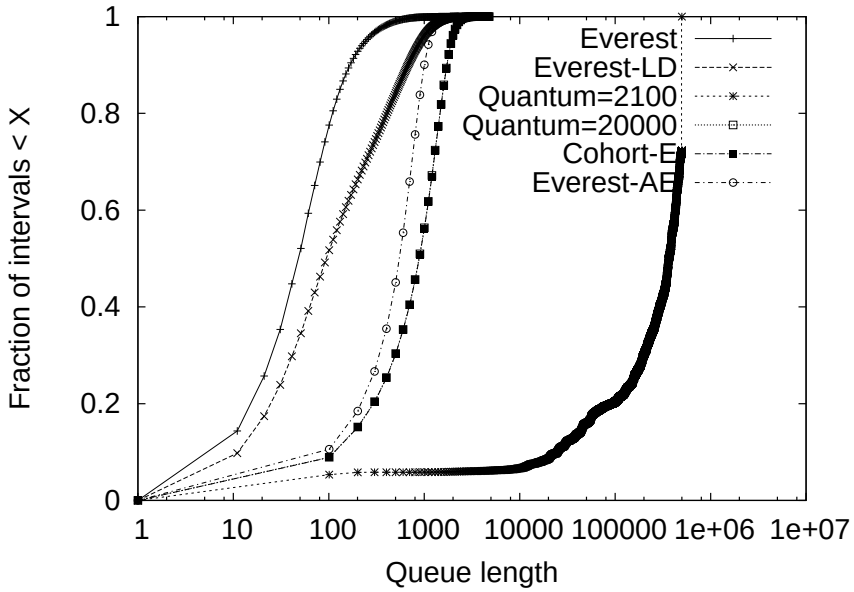


Figure 3.4: Queue length distribution with different schedulers for $t_{sw} = 2$ ms. At large context switch overheads, work-conserving schedulers waste processing capacity due to unnecessary context switches, thereby leading to larger queue buildup.

E scheduler, and Everest-AE cause unnecessary context switches, thereby reducing the effective processing capacity of the system. This leads to larger queue buildup (as demonstrated by Figure 3.4) and result in a large number of packets suffering delays greater than the tolerance D. Benefits of such non-work-conservingness have also been observed with disk schedulers by Iyer et al [58]. Observe that Everest-AE performs better than Quantum and Cohort because of the allocation eligibility being based on the queue length; Everest-AE has more knowledge of when a packet may exceed its delay tolerance, and triggers processor allocation as soon as required. Also, observe in Figure 3.5 that the number of context switches with Cohort and Quantum do not change because these schedulers are not conscious of the delay tolerance (the parameter varied on x-axis).

Figure 3.6 shows the variation of the percentage of packets that observe delay greater than the tolerance ($D = 2.1ms$) in a system with different number of pro-

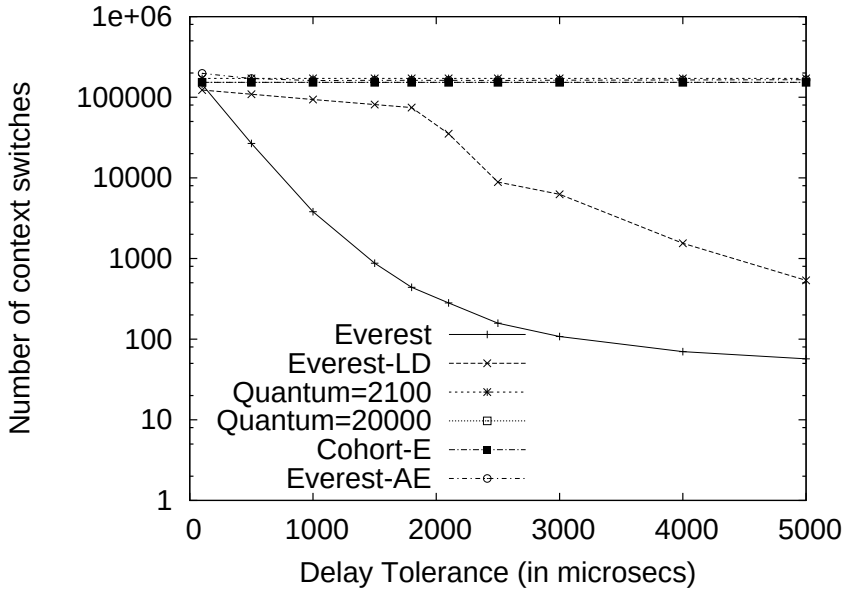


Figure 3.5: Number of context switches using each scheduler with increasing delay tolerance.

cessors (and with $t_{sw} = 2ms$). The graph shows that Everest outperforms the other schedulers by orders of magnitude at various levels of provisioning. At low levels of provisioning, all schedulers perform poorly (causing significant number of packets to exceed their delay tolerance). As the provisioning increases, the schedulers get more leverage to multiplex processors. However, work-conserving schedulers eagerly switch processors, thereby reducing the effective processing capacity significantly. Everest, on the other hand, switches a processor only when required, thereby decreasing the packets exceeding their delay tolerance as provisioning increases. The difference between Everest and Everest-LD demonstrates the benefit of agility at various levels of provisioning. Notice that Everest-LD can perform worse than Static because with Everest-LD, an aggregate that releases processors for other aggregates will not be able to reclaim the processors till some of its packets exceed their delay tolerance.

Figure 3.7 shows the cumulative distribution of the times for which each

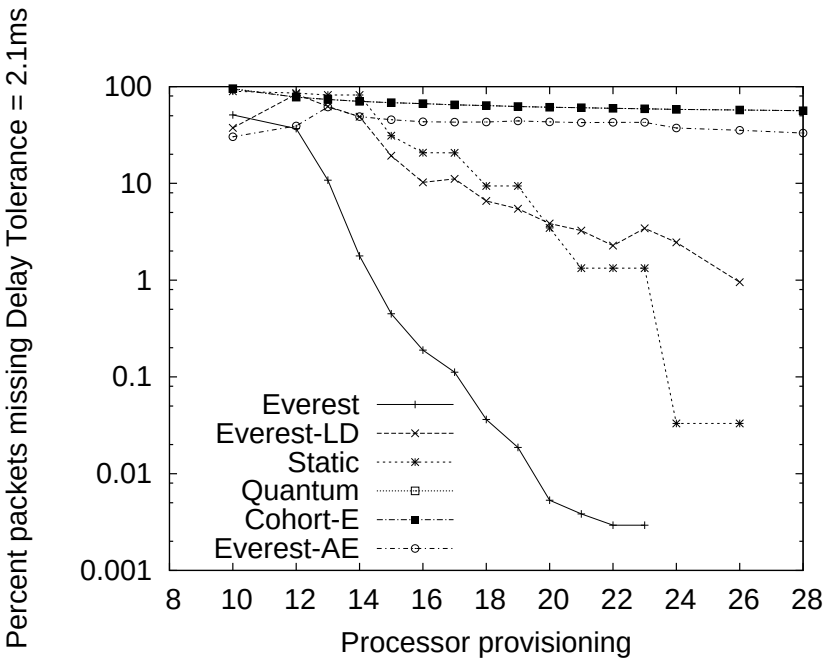


Figure 3.6: Effectiveness of Everest in reducing the number of packets that miss their delay bounds with increasing processor provisioning in the system.

processor is allocated to an aggregate before switching to a different aggregate when using Everest. During the time a processor is allocated to an aggregate, it could be actively processing packets or idle waiting for packets. This graph demonstrates the variable-quantum property of Everest; it adaptively chooses the right timescale for adapting processor allocations to minimize the number of context switches, while maximizing the number of packets that are served within their delay tolerance. The larger the processor provisioning level, the greater is the length of time a processor remains allocated to a single aggregate, and hence the smaller is the number of context switches.

In summary, the results indicate that the two properties—agility and wariness—can help Everest reduce the number of packets that exceed their delay tolerance by orders of magnitude where possible.

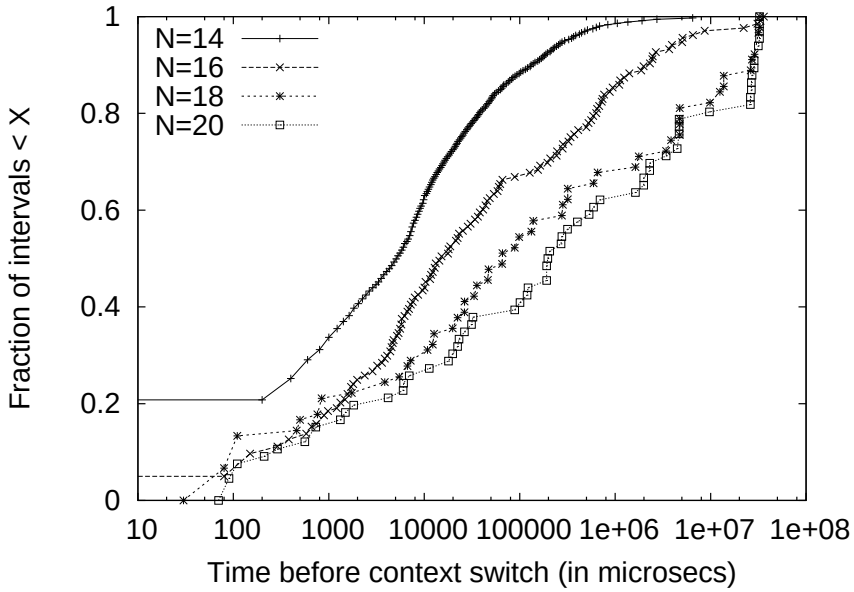


Figure 3.7: Variable quantum property of Everest.

3.6.3 Design space exploration

The principles on which Everest is designed are motivated by the relationship between the constraints of state-of-the-art programmable hardware, and the characteristics of network services—the delay tolerance of packets is of the same order as the context switch overhead ($D \sim t_{sw}$), and the processing time per packet is much smaller than the context switch overhead ($t_{sw} \gg t_{pkt}$). Consequently, Everest is a *point-solution* for systems with the above relationship between the characteristics. To broaden the scope of the scheduling problem and understand the behavior of the schedulers for virtualized routers better, in this section, we study the performance of different schedulers to varying relationships between the system and application parameters D, t_{pkt} and t_{sw}. To make the experiments more tractable, we hold t_{pkt} constant and only vary D and t_{sw}.

Figure 3.8 compares the percentage of packets that suffer delays greater than the delay tolerance with varying ratio of t_{sw} and D when using different schedulers.

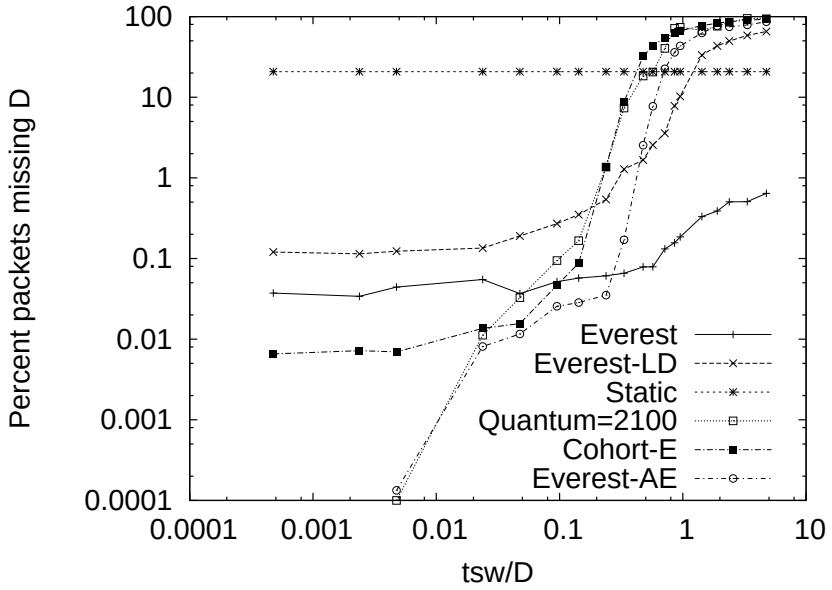
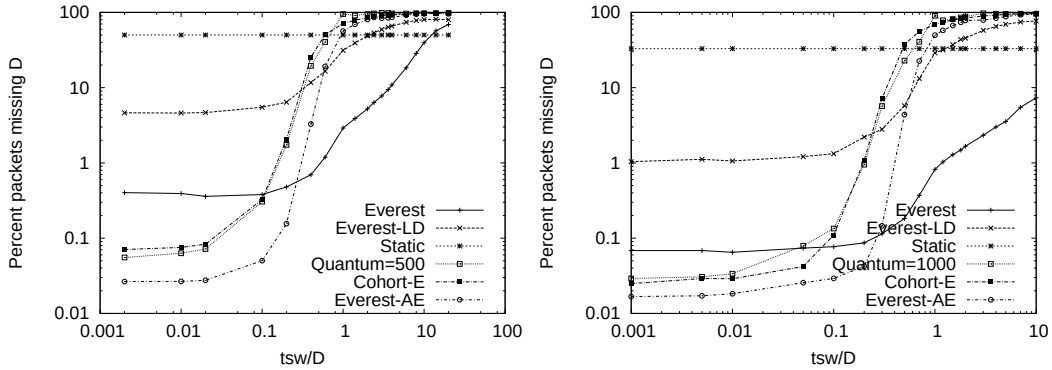


Figure 3.8: Behavior of schedulers with varying relationship between D and t_{sw}. We set $D = 2.1ms$.

The system contains 16 processors, D is set to 2.1ms, and the workload used is the UNC-2.5 trace, and t_{sw} is varied. The graph shows that Everest provides orders of magnitude better performance than conventional schedulers at large context switch overheads ($t_{sw} > 0.5 * D$), as expected. A system that statically allocates processors is unaffected by context switch overhead. Figure 3.9 shows that the same observations can be made when D is fixed at 500μs and 1 ms. As t_{sw} increases, the gap between Everest and Everest-LD increases, demonstrating the benefit of agility.

At smaller values of t_{sw}, work-conserving schedulers (Quantum, Cohort and Everest-AE) perform better; making all aggregates with packets to process eligible to receive additional processors, enables the schedulers to keep smaller queues by paying small context switch overhead. Figure 3.10 demonstrates this phenomenon; it plots the distribution of queue lengths for various schedulers.

As t_{sw} increases, work-conserving schedulers end-up with longer queues because of eager switching of processors (Figure 3.4). Also, the difference between the



(a) D = 500μs (b) D = 1000 μs

Figure 3.9: Behavior of schedulers with varying relationship between D and t_{sw} for various values of D. The graphs demonstrate that the behavior of the schedulers is similar at different values of D.

work-conserving schedulers and Everest at large context switch overheads demonstrates the benefit of wariness. With increasing t_{sw}, the amount of time spent in context switching vs. processing packets increases. Figure 3.11 shows the ratio of time spent context switching and the time spent processing packets with increasing context switch overhead for different schedulers. The ratio is high for work-conserving schedulers—Quantum and Cohort only at large context switch overheads. At large t_{sw}, minimizing context switches enables us to minimize reduction in the effective processing capacity.

While Everest detects the possibility of a packet missing its delay tolerance early and allocates processors accordingly, the graph in Figure 3.8 shows that several packets still miss their delay tolerance; the number of packets missing their tolerance increases with the context switch overhead. To understand this behavior of Everest better, we plot in Figure 3.12 the cumulative distribution of the amounts of time the request for a processor is denied (because it is allocated to another aggregate) before being satisfied. The graph shows that even with a small context

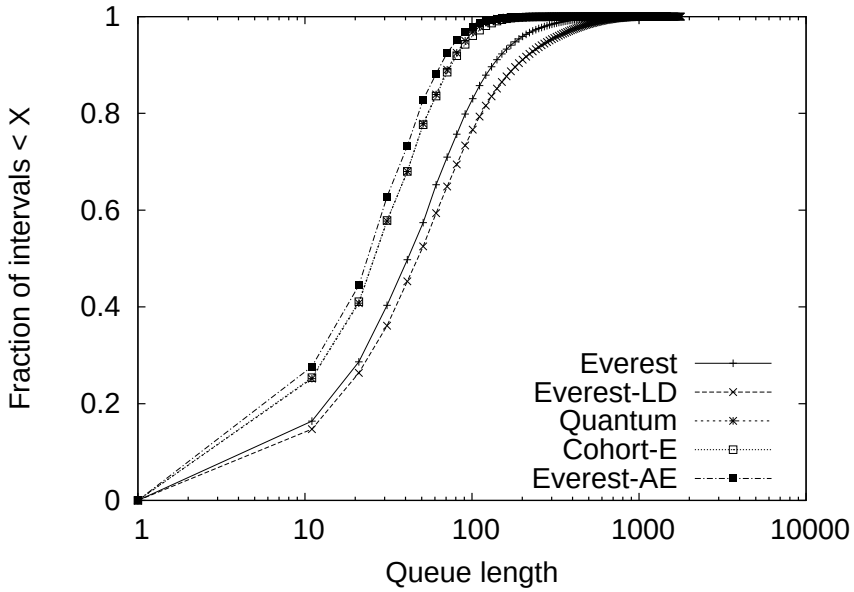


Figure 3.10: Queue length distribution with different schedulers at $t_{sw} = 1\mu$s. At small context switch overheads, work-conserving schedulers switch processors eagerly (and pay a small cost to avoid processors from idling when there are packets to process) and thereby manage to maintain smaller queues at aggregates.

switch overhead of 1μs (i.e., when $\frac{t_{sw}}{D} = 0.005$), there exist instances where time of denial is greater than 10 ms. With increasing context switch overhead, the time of denial increases. Also, the total time for which processor allocation requests were denied increases with t_{sw}. Further, observe that when $D < t_{pkt} + t_{sw}$, even packets that trigger new processor allocations will miss their delay tolerance (Equation 3.6). The effect can be observed as a steeper increase in packets exceeding their delay tolerance with Everest in Figure 3.8.

Figure 3.13 compares the different schedulers with respect to the average delay observed for packets in systems with different values of $\frac{t_{sw}}{D}$ at $D = 2.1$ ms. It demonstrates that by minimizing the total number of context-switches, Everest, a non-work-conserving scheduler, achieves an order of magnitude smaller average delay per packet as compared to the Quantum, Cohort, and Everest-AE schedulers.

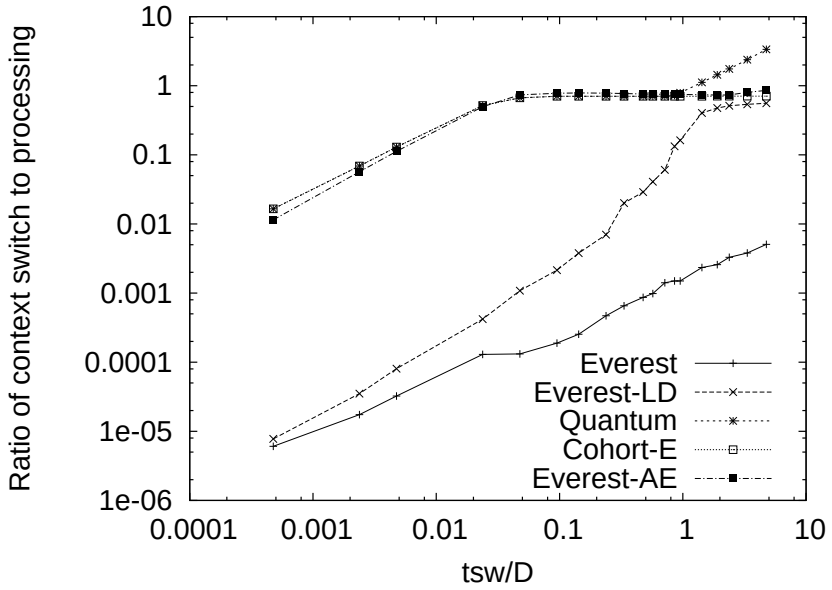


Figure 3.11: Effect of increasing context switch overhead on the processing capacity on different schedulers. Work-conserving schedulers suffer substantial loss in processing capacity, the effect of which is considerable at large context switch overheads.

Further, the graph demonstrates the effectiveness of Everest even in systems with much smaller context-switch times ($0.05 < \frac{t_{sw}}{D} < 1$) as compared to today's state-of-the-art.

Figure 3.14 demonstrates that context switch overhead plays a significant role in the non-work-conservingness of Everest. Recall that Everest marks processors for release when $R_{arr}(\Delta) > R_{dep}(\Delta)$. In all the previous experiments, we chose $\Delta = t_{sw}$. In this graph, we experiment with different values of Δ. The graph shows that when the context switch overhead is small, choosing a large value of Δ causes processors to idle for longer duration even though there are eligible aggregates. At large value of context switch overhead, choosing a small value of Δ causes processors to be released eagerly; increased number of context switches reduces the effective processing capacity, thereby increasing the number of packets that exceed their delay tolerance. Since different processor release latencies are good for systems

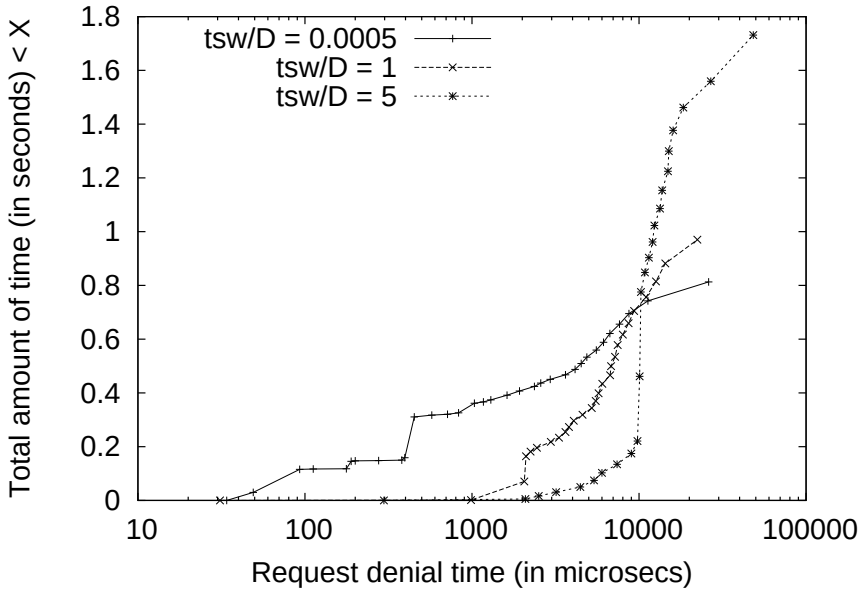


Figure 3.12: Contention CDF for Everest.

with different context switch overheads, the graph demonstrates the effectiveness of choosing $\Delta = t_{sw}$.

Finally, we validate our conclusions about the behavior of the schedulers under different workload conditions by using several traces with different characteristics as shown in Table 3.5. Figure 3.15(a) and 3.15(b) plot the percentage of packets that suffer delays greater than the aggregate's tolerance for different UNC traces collected at different times with varying $\frac{t_{sw}}{D}$. Figure 3.16(a) and 3.16(b) plot the percentage of packets that suffer delays greater than the aggregate's tolerance for traces of different peak bandwidths (1 Gbps and 10 Gbps) with varying $\frac{t_{sw}}{D}$. All these graphs demonstrate the same observations that we made in Figure 3.8. Further, observe that at various values of $\frac{t_{sw}}{D}$, one of Everest andEverest-AE outperform all other schedulers in minimizing the number of packets that exceed their delay tolerance.

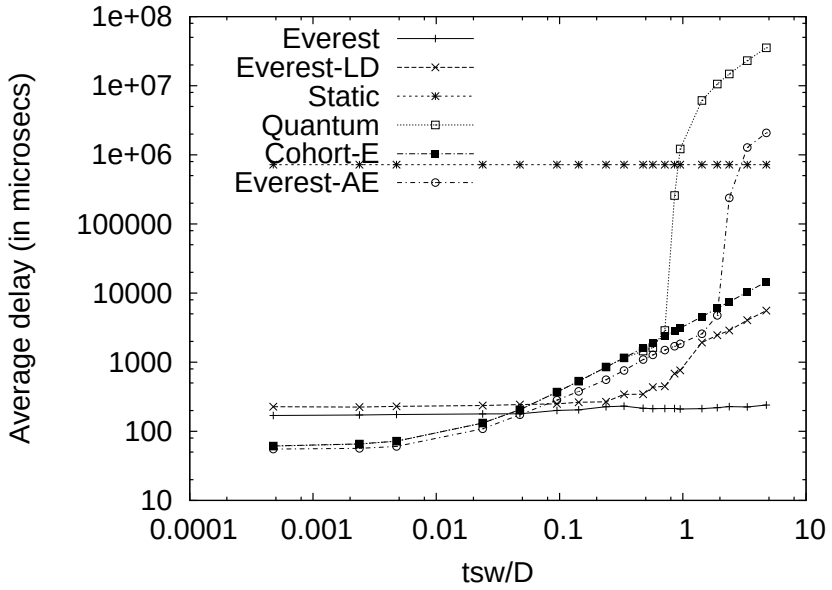


Figure 3.13: Average packet delay with different schedulers. Non-work-conserving schedulers cause longer packet delays at small context switch overheads. At large context switch overheads, a wary scheduler like Everest can minimize the wastage of processing capacity in context switching, thereby incurring lower average packet delay.

3.6.4 Scheduling Framework for Multi-processor Virtualized Routers

Figure 3.17 summarizes the behavior of Everest and Everest-AE at various values of $\frac{t_{sw}}{D}$ for traces collected at different times, traces of different bandwidths, and for different delay tolerance values. The graph compares the ratio of packets that miss their delay tolerance with Everest-AE and Everest. A value less than one represents that Everest-AE performs better than Everest, and a value greater than one represents that Everest is better. The graph shows that for different settings, Everest clearly performs better than Everest-AE when $\frac{t_{sw}}{D} \geq 1$ and Everest-AE performs better when $\frac{t_{sw}}{D} \leq 0.1$.

When $\frac{t_{sw}}{D} \in (0.1, 1)$, the steep increase in the lines shows that a small variation of $\frac{t_{sw}}{D}$ can make either Everest or Everest-AE better than the other by orders of

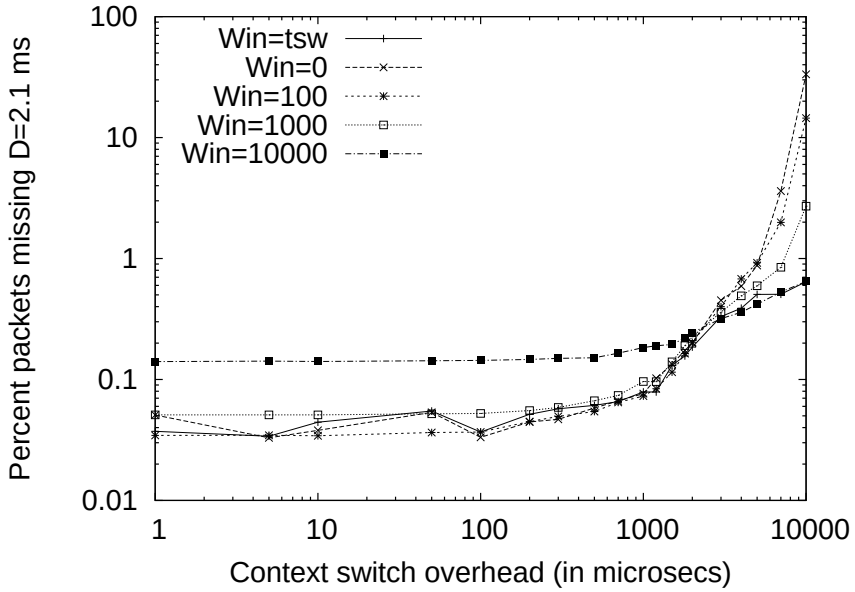
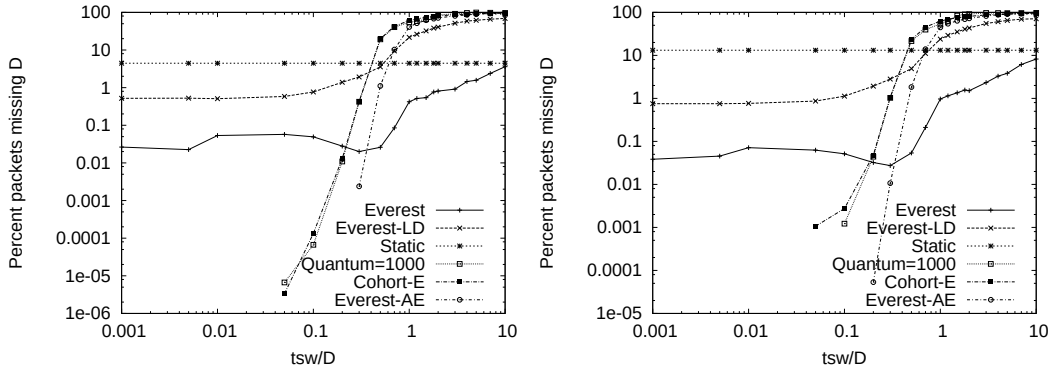


Figure 3.14: Effect of release (voluntary) latency. The values of Win in the legend are in microseconds.

magnitude. Further, observe the difference between the lines for UNC-51, UNC-58 and others. For UNC-51 and UNC-58 with small values of t_{sw}, the provisioning ($N_p = 16$) is sufficient for Everest-AE to process all packets before their delay tolerance. However, for other traces, the provisioning is not sufficient even at very small values of t_{sw} to process all packets before D. These observations lead us to conclude that it is challenging to exactly determine the transition point without knowing the workload pattern, and its interaction with other parameters like N_p, D, t_{sw} and t_{pkt}. Further, since workloads vary with time, one scheduler may perform better than the other at different times; drawing conclusions based on a trace of packets can be incorrect.

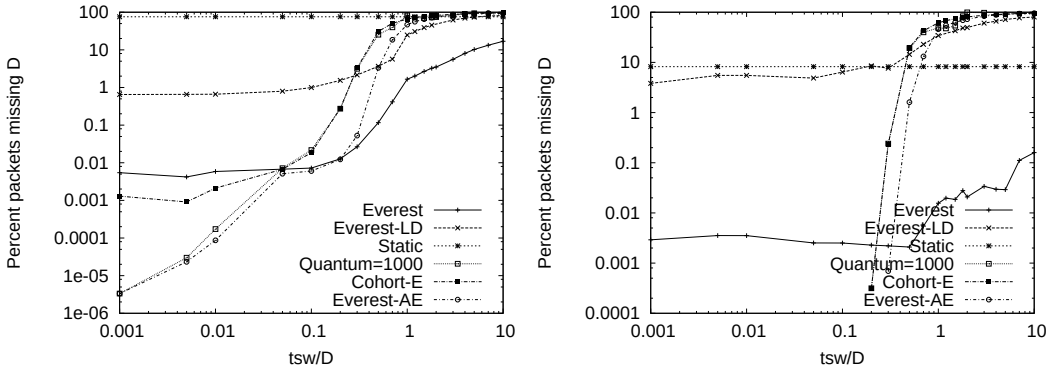
One way to address the problem of picking the right scheduler is for the system to *learn* by observations during system operation, which among Everest and Everest-AE performs better for the current parameter settings and the workload pattern, and use the better scheduler. Another option is to pick a delay tolerance



(a) UNC-25-0830 (b) UNC-25–1100

Figure 3.15: Behavior of schedulers with traces collected at different times and different days. Quantum size is in microseconds.

value (since this value is configurable by the network service provider) that is either substantially larger than t_{sw} and use Everest-AE, or choose a D comparable to or smaller than t_{sw} and use Everest.



(a) b/w = 1Gbps, $N_p = 10$ (b) b/w = 10Gbps, $N_p = 48$

Figure 3.16: Behavior of schedulers with different system and workload parameter settings. Quantum size is in microseconds.

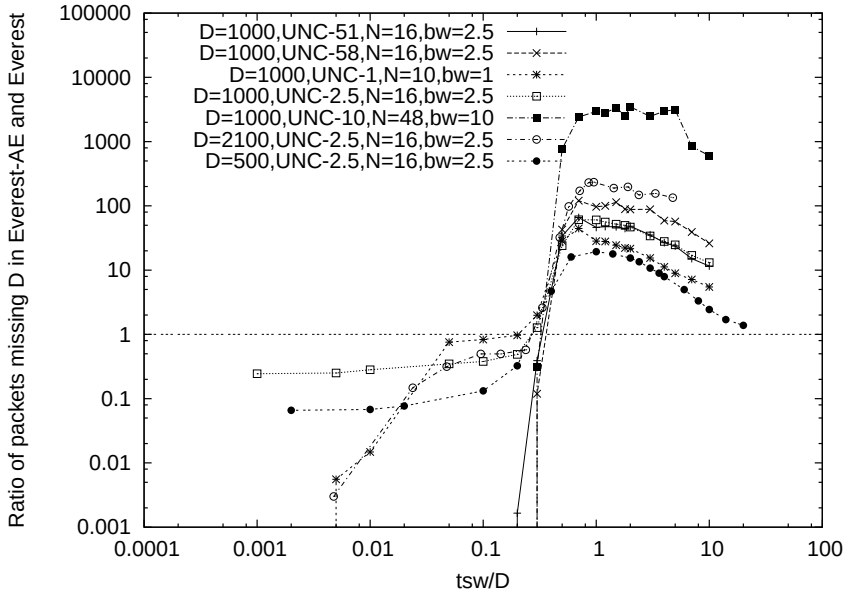


Figure 3.17: Summary of the behavior of different schedulers at various system, workload and application characteristics.

Chapter 4

ShaRE Prototype

In this chapter, we describe the implementation of a prototype of ShaRE on a platform consisting of a RadiSys ENP-2611 board [16] with an Intel's 600MHz IXP2400 network processor [57]. The board also contains three 1Gbps fiber ports for transmitting and receiving packets, a 128 MB SRAM, and a 1 GB DRAM. The IXP2400 network processor (Figure 4.1) includes: (1) eight RISC processor cores (optimized for fast path packet processing) called *micro-engines (ME)*, each with eight hardware threads, and an instruction store of size 4K instructions; (2) a general-purpose XScale® processor; (3) a multi-level memory hierarchy consisting of memory local to individual micro-engines (of size 2.5KB), on-chip scratchpad memory (of size 16 KB) shared across all micro-engines, and interfaces to external SRAM and DRAM; and (4) next-neighbor rings—efficient channels for communicating packets between micro-engines. The board runs MontaVista Linux [11] on the XScale processor.

4.1 Design Overview

Baker and ShaRE instantiate the standard principle of hiding various resource implementations behind a common interface by exporting a packet-processing-oriented

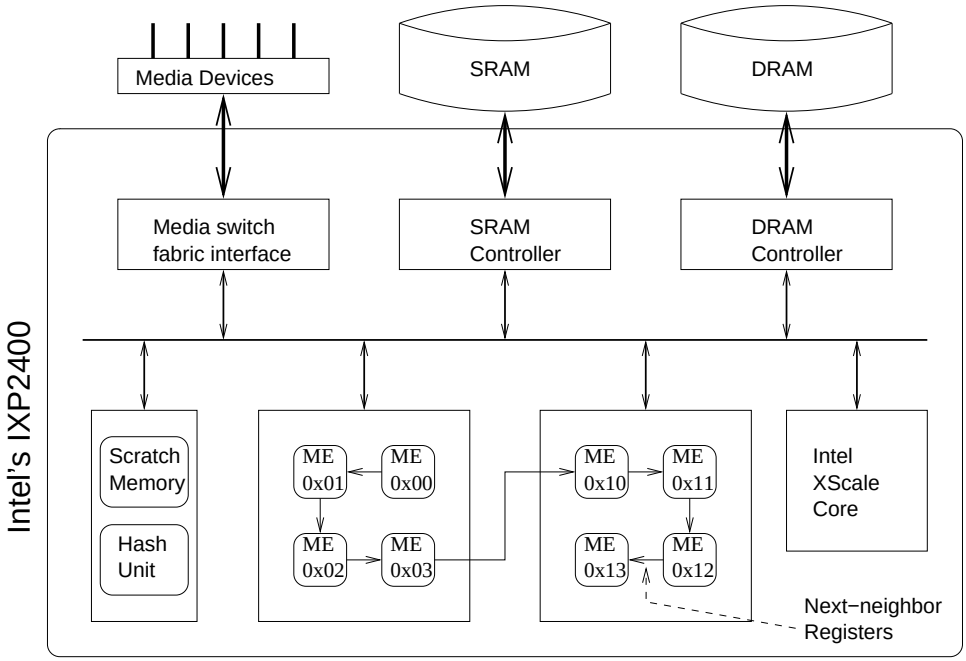


Figure 4.1: A simplified Intel's IXP2400 architecture (derived from [61]).

resource abstraction layer (pRAL). To accommodate diverse and complex high-performance router architectures, ShaRE supports multiple implementations of each pRAL interface. Binding pRAL interfaces to the appropriate implementations, however, faces several challenges in the domain of virtualized routers. First, to accommodate highly variable workload demands, ShaRE should allow the binding of interfaces to implementations to change at run-time. Second, the high rate of packet arrival and relatively short processing times of each packet make it important that ShaRE supports this flexible binding without adding much per-packet processing overheads. To meet both of these demands, we have developed an *adaptation-time linking* mechanism that rewrites the aggregate executables at adaptation-time to instantiate a specific set of resource implementations for pRAL interfaces.

To support dynamic adaptation of resource allocations, ShaRE must also support mechanisms to *replicate*, *checkpoint* and *migrate* aggregates. The high rate of

packet arrival and relatively short processing times of each packet require that such a check-pointing and state migration mechanism be lightweight. ShaRE exploits the regular structure of network services to instantiate an *application-aware checkpoint and migration* mechanism that suspends, replicates, or migrates an aggregate only when it is safe to do so (e.g., when it is not holding a lock) and when it is inexpensive to do so (e.g., when state is minimal).

In the rest of the chapter, we make this discussion concrete by describing the challenges and our solutions in implementing these mechanisms.

4.2 pRAL

ShaRE exploits several characteristics of high-performance virtualized routers to define *packet-processing oriented* resource abstractions—(1) the programmable hardware used to build the routers can contain heterogeneous processing elements that are optimized for specific tasks, (2) the hardware can support different types of inter-processor communication channels to efficiently support pipelined applications (in which a packet gets processed by several aggregates arranged in a pipeline before getting transmitted), (3) processing of packets that share application state can be done in parallel (and hence accesses to the state should be synchronized), and (4) several packet processing functions that are executed often (e.g., hash, crypto, lookup) can be implemented by fixed function elements in the hardware.

The prototype's pRAL interface exports interfaces for four different types of hardware resources: (1) *processing units*, (2) *inter-processor communication channels*, (3) *processor synchronization mechanisms*, and (4) *fixed-function* accelerators. The interface for processing units includes methods to load, start, stop, and checkpoint the unit. The interface for inter-processor communication channels include methods to enqueue and dequeue packets into a packet channel, measure the queue length, and arrival and departure rate of packets. The synchronization interface sup-

ports locks, with methods to acquire and release the locks. And, the fixed-function interfaces include abstractions provided by hardware accelerators such as hash unit and crypto unit. Each hardware accelerator interface supports an *eval* method.

Our prototype supports several implementations for each of these interfaces. For the processing unit, we support micro-engines (MEs) and the XScale®. MEs, with support for multiple hardware threads, can achieve high packet-processing throughput; however, they offer a much constrained execution environment (e.g., no data or instruction caches and no support for floating-point operations). Conversely, the XScale® is a flexible, general-purpose processor, but can only support low throughput.

For the inter-processor communication interface, the prototype supports: (1) next-neighbor rings[1], (2) scratchpad rings, and (3) SRAM rings. While next-neighbor rings can be used to communicate between adjacent MEs, scratchpad and SRAM rings can be used to communicate across any pair of MEs. The scratchpad and SRAM rings consume global shared resources (scratchpad rings use on-chip bus bandwidth and compete for scratchpad space, whereas SRAM rings use both on-chip and off-chip bus bandwidth). As a result, it is desirable to use next-neighbor rings when possible, otherwise to use scratchpad rings if sufficient scratchpad space is free, and to fall back on SRAM rings when necessary.

For the synchronization interface, we support two implementations: (1) *local-locks* implemented in memory local to each ME that can be used to synchronize threads running within an ME, and (2) *global-locks* implemented in on-chip scratchpad that can synchronize threads running on different MEs. Whereas local-locks can be accessed within a single cycle, accesses to global-lock incur an overhead of at least 60 cycles and consume globally-shared scratchpad memory bandwidth [61]. Hence, unless an aggregate is replicated on multiple microengines, using local-locks

[1]A ring is a circular FIFO buffer than can be used for storing packet handles [61].

to synchronize threads provides better performance.

For the fixed function interfaces on the IXP2400, we currently only support hardware implementation of 128-bit hashing. The implementation of other fixed functions is straight-forward.

4.3 Adaptation-time linking

Since each pRAL interface may support multiple implementations, each usage of a pRAL interface has to be bound to the best-suited resource implementation. Binding the most efficient implementation at *compile-time* is inflexible and limits any run-time adaptation of resource allocations. Binding the most flexible resource implementation to each interface, on the other hand, can be very inefficient. For instance, binding the next-neighbor implementation between two communicating aggregates precludes the possibility of replicating the aggregates in response to traffic fluctuations. Whereas, binding the SRAM implementation between the aggregates can hurt the throughput of the system substantially. To illustrate, Figure 4.2 shows the effect of using different channel implementations for communicating packets between two aggregates of an IPv4 forwarding network service. The graph shows that using the most efficient implementation whenever possible is critical to maximize the throughput of a virtualized router.

The goals of adaptation flexibility and high-performance can be met simultaneously through run-time binding. At compile-time, we can include all possible resource implementations in the aggregate executables, and perform a run-time check to identify the appropriate resource implementation (e.g., through appropriate initialization of function pointers). When ShaRE adapts allocation of resources to aggregates, the binding can be changed simply by re-initializing the function pointers for each usage.

Unfortunately, implementing such run-time binding on our platform poses

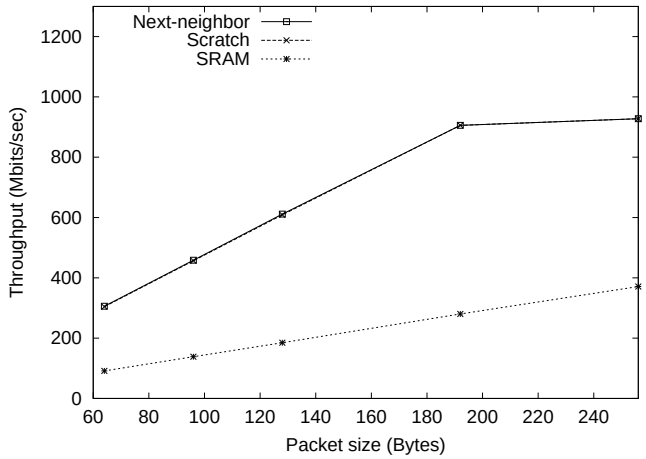


Figure 4.2: Benchmark showing the effect of using different channel implementations on throughput of an IPv4 forwarding application. Observe that the next-neighbor and scratch implementations do not differ in throughput because of the presence of multiple threads per processor core; the eight threads on each microengine are sufficient to mask the latency incurred in accessing a scratch channel, but are not sufficient to mask the latency of an SRAM channel.

two problems. First, resolving a function pointer involves memory accesses; the run-time overhead of such memory accesses for each packet reduces the sustainable packet throughput. Second, micro-engines on IXP2400 support an *instruction store* (and not a instruction cache)[2]; hence, the entire code for the aggregate running on an ME must reside completely in the instruction store. The run-time binding mechanism requires that the code for all the possible pRAL implementations that an aggregate may be bound to must be resident in the code store. The limited code store size of 4K instructions makes this infeasible.

We resolve the two problems and support dynamic binding of pRAL interfaces to implementations by implementing an *adaptation-time linking* mechanism in ShaRE. The linker takes as input (1) the binary for an aggregate containing un-

[2]An instruction can be written into the instruction cache from a higher level memory while the microengine is executing other instructions. In contrast, in order to write an instruction into an instruction store, the microengine has to be stopped.

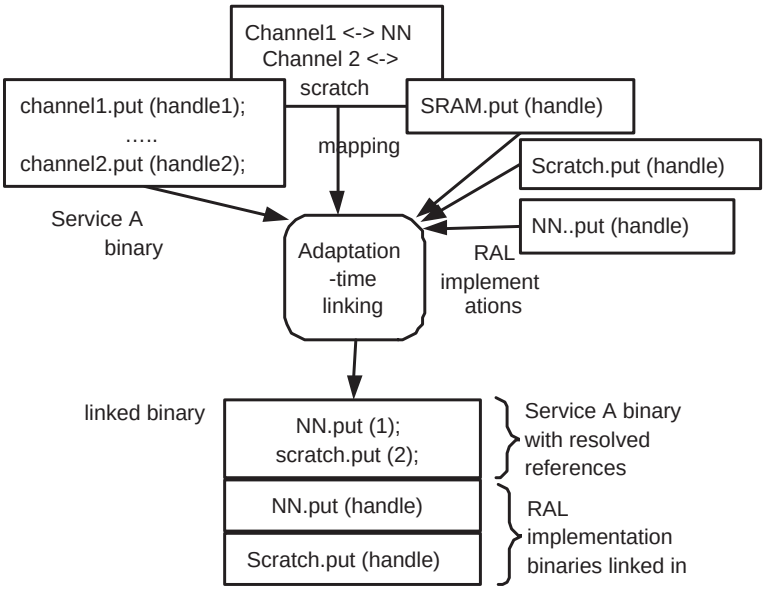


Figure 4.3: Adaptation time linking.

resolved references to pRAL methods, (2) a list of mappings from pRAL interface instances to implementations, and (3) the binaries for pRAL implementations (see Figure 4.3). The linker substitutes all of the pRAL method calls in the aggregate binary with calls to the appropriate methods of specific implementations. Further, since the micro-engines do not have hardware support for relative addressing, the linker relocates the pRAL implementation binary explicitly. The linker is invoked during initialization when an aggregate is first loaded, and every time the binding of an aggregate's resources is changed. This adaptation-time linking does not impose any run-time overhead once the adaptation is complete, and minimizes the instruction store requirements for storing the pRAL implementations (since only the required implementations are linked in the aggregate binary).

4.4 Check-pointing and State Migration

Rebinding resource implementations requires check-pointing and migrating aggregate's state—code and data—from one resource to another. In high-throughput environments like virtualized routers, it is crucial to minimize the impact of adaptation on system performance. Whereas the techniques for check-pointing and migrating a process's state are well-studied [53, 72, 75, 78, 86], designing a lightweight check-pointing and state migration mechanism for high-performance routers is challenging due to high rate of packet arrivals.

ShaRE exploits the characteristics of aggregates to minimize the check-pointing overhead. Observe that aggregates are organized in a loop that performs three steps.

```
while (1) {
    dequeue a packet from an input channel;
    process the packet;
    enqueue the packet on output channel;
}
```

At the beginning of this loop, the aggregate contains no transient state (e.g., stack); further, the aggregate does not occupy any lock. Thus, halting processing units only at the beginning of loops makes the process of checkpointing and state migration both *safe* and *efficient*.

To support such application-aware check-pointing, each thread running on an ME can check at the top of the loop some memory location to determine whether or not to halt; however, this introduces memory access overhead while processing each packet. To eliminate this overhead, our implementation delivers the halt command to a micro-engine by: (1) stopping the micro-engine; (2) writing a thread-halt instruction explicitly in the instruction store (by replacing an appropriately placed no-op instruction at the top of the loop); and (3) restarting the micro-engine. This

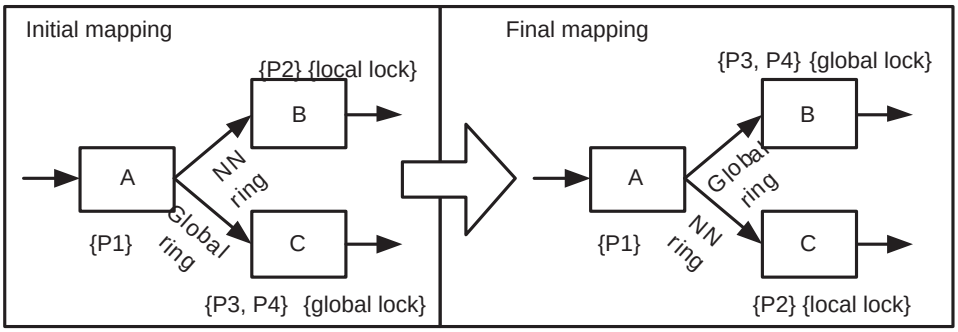


Figure 4.4: Adaptation example.

ensures that each hardware thread on the micro-engine will eventually stop at the halt instruction. We use polling to determine when all the threads have reached the safe halting state.

Once all the threads on a micro-engine halt, the state migration mechanism migrates any state (packet handles) present in the communication channels. This is accomplished using the enqueue and dequeue methods exported by the individual channel implementations. This approach requires both the producer and consumer MEs to be halted prior to migrating the channel state. An alternate approach is to simply wait until the consumer drains the channel completely; this eliminates the need for channel state migration but can increase the adaptation latency. We do not consider the alternate approach in our current prototype.

4.5 Adaptation Example

To illustrate the invocation of the mechanisms during adaptation, consider a system with four micro-engines ($P1$-$P4$) and three aggregates (A, B, and C)(see Figure 4.4). Let the initial processor mapping be: $A \rightarrow P1$, $B \rightarrow P2$, and $C \rightarrow \{P3, P4\}$. Since A and B are mapped on adjacent micro-engines, they communicate using the next-neighbor ring channel implementation, while packet communication from A to

C uses the global scratchpad-ring implementation. In this arrangement, the lock interface used in B is mapped to the local-lock implementation (that can synchronize only the threads within a micro-engine), while the lock usage in C is mapped to a global-lock implementation.

Now, consider the case that, because of fluctuations in traffic, B needs to be allocated 2 micro-engines and C's allocation can be reduced to one micro-engine. In this case, to allow the communication from A to C to be carried over next-neighbor rings, the allocation of processors should be adapted to: $A \rightarrow P1$, $B \rightarrow \{P3, P4\}$, and $C \rightarrow P2$. This requires the channel from A to B to be re-bound to scratchpad-ring implementation, whereas the channel from A to C to be re-bound to next-neighbor ring implementation. Further, the lock implementations in B and C need to be bound to global- and local-locks, respectively. ShaRE takes the following steps to transition the system into this new state of resource allocations.

1. Checkpoint processor $P1$ through $P4$.

2. Migrate packet handles (1) from the next-neighbor ring implementation connecting A and B to a new scratchpad ring implementation; and (2) from the scratchpad ring implementation connecting A and C to the next-neighbor ring implementation.

3. Bind the channel usage for the A to B and A to C communication to scratchpad ring and next-neighbor ring, respectively. Bind the lock usage in B to use global lock, and the one in C to use local lock implementations.

4. Load the new binaries on $P1$-$P4$, and restart the micro-engines.

4.6 Implementation Details

ShaRE has been implemented in collaboration with Intel's Communications Archi-

tecture Lab; much of the implementation of the mechanisms for adaptation-time linking, and checkpointing and state migration was done by our collaborators at Intel. Overall, ShaRE contains about 50,000 lines of C/C++ code, and is compiled and run as a set of loadable modules on the XScale core. Additionally, we have built a command line interface to ShaRE that runs as a user-level program on the XScale core, and is accessible through the serial-line debug port on the ENP-2611 board. ShaRE uses several functions provided by the Intel's IXA SDK Hardware Abstraction Layer [7] to access the various resources on the IXP2400.

The Everest scheduler runs on the XScale core as a user level thread. During initialization, ShaRE provides Everest with a list of packet channel handles. Everest can access the methods supported by packet channel interfaces (like current queue length, total packets received since the count was last reset, etc.). Using these methods, Everest monitors the queue lengths, estimates R_{arr} and t_{pkt}, and adapts processor allocations using the mechanisms provided by ShaRE when a queue length crosses a threshold.

4.7 Microbenchmark Results

4.7.1 Effect of pRAL on throughput

Since pRAL methods are linked in as function calls, they incur an overhead of 20 cycles for each invocation; this overhead accounts for the latency introduced by branch, return and stack setup operations. The impact of this overhead on packet throughput depends on the aggregate (on the number of pRAL method calls included in the aggregate, and the number of cycles required to process a packet without this overhead). We measure this impact for an aggregate of the IPv4 packet forwarding service by considering two scenarios, one using pRAL methods as function calls, and the other with in-lined pRAL methods. The aggregate has two pRAL calls, one for

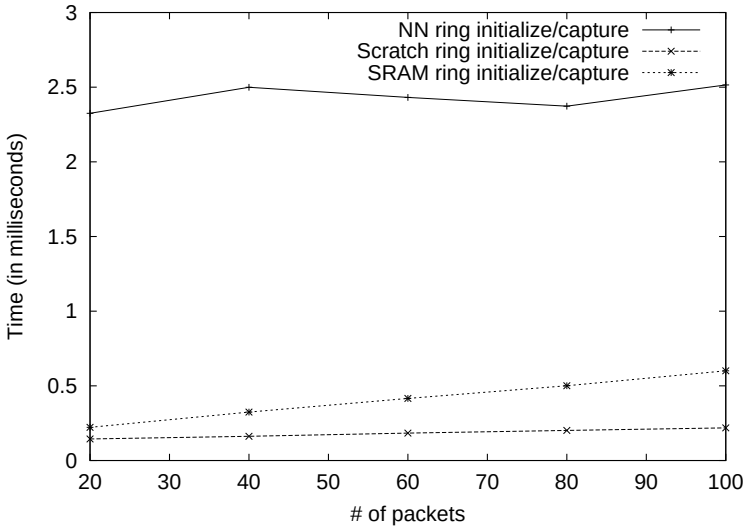


Figure 4.5: Overhead of channel state migration.

dequeuing and one for enqueuing packets. In this case, the measured throughput supported by a single micro-engine dropped by 14%, from 388Mbps to 335Mbps (assuming 64-byte packets). This result gets validated by our observation that the aggregate with no pRAL overhead (i.e., inlined pRAL calls) takes 250 cycles to process a packet. Hence, an overhead of 40 cycles should lead to a 14% degradation of throughput.

4.7.2 Check-pointing and state migration overhead

The latency for halting/check-pointing a micro-engine in a safe state has three components: (1) a 60μs overhead to inform the micro-engine to stop at the beginning of the loop, (2) a 34μs overhead to check that the micro-engine has actually stopped[3], and (3) the time taken by all threads in the micro-engine to reach the beginning of

[3]On the Intel's IXP2400, the only way to check whether a micro-engine has stopped is to poll the micro-engine continuously to check if the program counter has actually reached the halt instruction, hence this step is required.

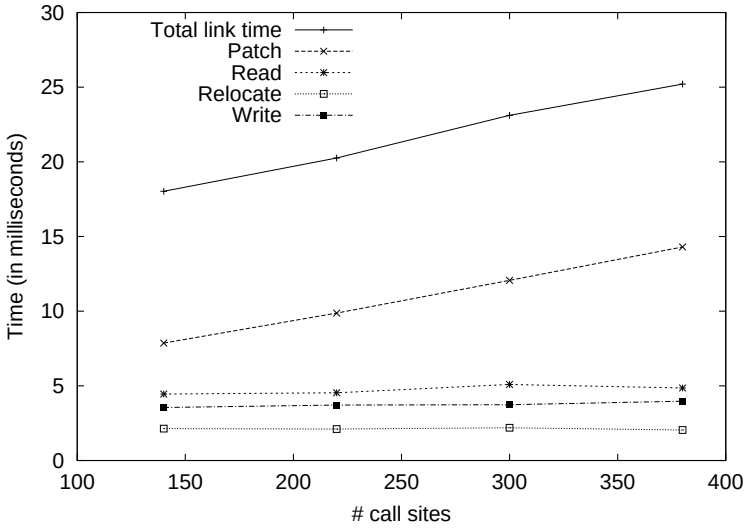


Figure 4.6: Overhead of adaptation-time linking.

the loop, which is at most equal to the time to process a packet.

Figure 4.5 shows the overhead of packet channel migration for the different channel implementations. The graph shows that the overhead of reading/restoring packet handles from/to different channel implementations is proportional to the number of packet handles, as expected. Migrating packet handles from next-neighbor ring incurs substantially higher overhead than the others because it requires loading and executing code on a micro-engine that saves or initializes the next-neighbor packet channel; the XScale core cannot access the next neighbor rings directly. In fact, this observation exposes a tradeoff between (1) migrating packets from/to a next-neighbor channel and (2) letting the packet channel drain and avoid migration. For the scratchpad and SRAM ring implementations, the XScale® core can directly migrate packet handles; hence, the overhead is lower.

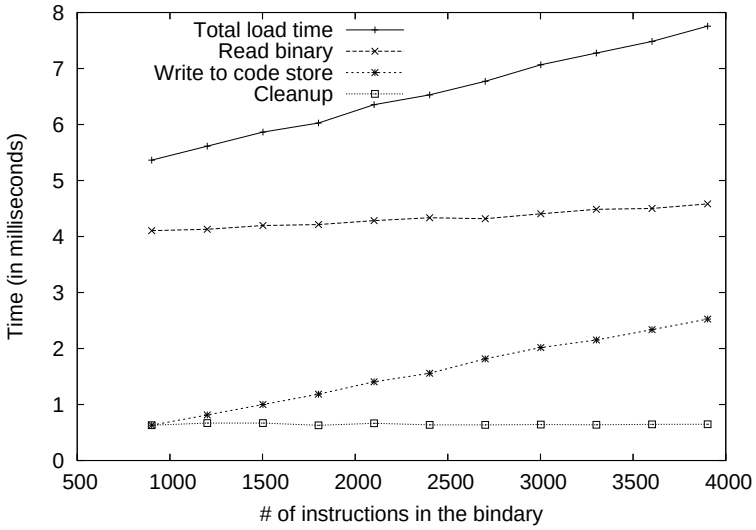


Figure 4.7: Overhead of loading.

4.7.3 Binding

Binding involves multiple steps: reading the binary from a memory-mapped file system on the XScale® core, relocating pRAL implementations, patching all the *call sites* (locations of RAL calls in the code), and writing the linked binary in the file system. Figure 4.6 shows the binding overhead as a function of the number of call sites. The total time varies linearly with the number of call sites. Linking also incurs a fixed overhead for reading a binary from the file system, writing the linked binary to the file system, and relocation of a packet channel and a lock implementation.

4.7.4 Loading and starting micro-engine

Figure 4.7 shows the overhead to load the code store of a micro-engine. This overhead has three components: a fixed overhead for reading the binary from the file system, writing the binary to the code store, and a fixed overhead to free-up resource allocated in reading the binary. The overhead to write the binary into the

micro-engine instruction store is proportional to the number of instructions. Once the code store is loaded, it takes 36μs to start the micro-engine. We make two observations here— (1) if the linked binary is cached after reading, we do not incur the overhead of reading it from the file-system when required subsequently, and (2) the overhead of cleanup does not contribute to the adaptation overhead. Hence, the contribution of the loading mechanism to adaptation latency is about that required for writing into the code store, which is at most 2.5 ms.

4.8 Discussion

The results in Section 4.7 show that the binding mechanism contributes a significant percent of the total adaptation latency. Consider, for instance, a design point where an aggregate deployed on an ME has 200 call sites, and has 2000 instructions, and the associated packet channel has 100 packets in a scratch ring that needs to be adapted to a next-neighbor ring. For this case, adapting the ME takes around $25ms$. Of this time, the time required to stop the ME, load the ME with the new code, and re-start the ME is about $1.5ms$; the remaining time is contributed by the binding mechanism.

The overhead of the binding mechanism affects (1) the duration for which a particular processor is unavailable for processing packets (and hence the amount of packet loss), and (2) the frequency with which processor allocations can be adapted. Observe that we can reduce the effect of binding overhead on processor down-time by overlapping the construction of the new binary image with packet processing; a processor can continue to process packets until the new binary image is ready to be loaded. Further, by caching the linked binaries for different resource instance to implementation mappings, we eliminate the linking overhead completely in the common case.

In summary, in the common case, we could reduce the overhead of adaptation

to be approximately equal to the overhead of writing into the code store, which is at most 2.5 ms.

The binding overhead exposes a fundamental trade-off between adaptation-time and run-time binding mechanisms. On the one hand, adaptation-time binding yields efficient code that imposes little run-time overhead and produces minimum size binary images; however, it incurs a significant adaptation latency. On the other hand, the use of run-time binding mechanism virtually eliminates the adaptation-time binding overhead; however, it may incur greater run-time overhead (per packet) and results in larger sized binary images.

While our prototype implements one solution to the binding mechanism (adaptation-time binding) that is motivated by the constraints placed by the platform at our disposal, the relative performance/applicability of these two binding mechanisms depends on different system characteristics and application requirements. For instance, in a system with hardware multi-threaded processors, the additional computational instructions resulting from run-time binding will have little impact on the packet processing throughput if the total number of computational instructions executed by all threads between successive memory accesses are already insufficient to hide memory access latency; in this case, addition of computation instructions only reduces processor stall and has little effect on the packet processing throughput. Similarly, which mechanism suits better depends on the relationship between the loss in throughput resulting from run-time binding, and the effect of reducing the frequency of adaptation and dropping packets (driven by application requirements) during adaptation with adaptation-time binding. Finally, run-time binding may be better suited for processors that support instruction caches (e.g., the Intel XScale® core on the IXP2400 network processor), have relatively large instruction stores (such that more than one resource instance could be pre-loaded), or process packets with a lower throughput requirement.

4.9 Related Work

State-of-the-art programming environments for programmable router hardware [5, 14, 21, 24, 50, 70, 87, 91, 102] allocate resources in the multiprocessor system to aggregates of packet processing services statically (at design time). Hence, none of the environments implement mechanisms required for run-time adaptation of resources. In this chapter, we develop a core set of mechanisms—checkpointing, state migration, and dynamic binding— required to adapt resources to aggregates at run time *safely* and *efficiently*.

Many past systems provide checkpointing and state migration [53, 72, 75, 78, 86], and dynamic binding mechanisms [45, 47, 80, 99] in the general-purpose application domain. However, a straight-forward application of the techniques could lead to a large unacceptable overhead that hurts the system throughput. To maintain high throughput, our ShaRE prototype exploits the unique characteristics of the domain of virtualized routers to achieve efficiency. By exploiting the loop nature of packet processing applications, we reduce the overhead of checkpointing and state-migration significantly. Similarly, given the requirement to support high throughput and the constraints on instruction store sizes, we explore the benefits and tradeoffs of using adaptation-time binding mechanism.

NetBind [70] and VERA [91] provide some mechanisms for dynamic allocation of resources in order to support extensibility of network services [36, 46, 64]. NetBind enables dynamic creation of packet processing pipelines through the dynamic binding of small pieces of machine language code. VERA focuses on making a router consisting of a PC with a host processor and a few network processors (1) extensible by allowing dynamic installation of new functionality, and (2) efficient by offloading the most frequently executed packet processing functions to network processors. Such offloading allows VERA to support router extensions on the host processor. However, neither NetBind nor VERA contain mechanisms to adapt pro-

cessor allocations at run-time to maximize the throughput of the system in the presence of traffic fluctuations.

Chapter 5

Network Services Case Study

5.1 Setup

To demonstrate the effectiveness of ShaRE, we have implemented a high-performance virtualized router that hosts several services, including an L3-switch, a port-scan detector, a TCP-SYN flood detector, and a worm signature matcher. These services form a crucial part of richer functionality that is desirable in today's Internet; recent attacks on the Internet have brought significant focus on research towards detection and control of unwanted traffic generated by worms, viruses, distributed denial of service (DDoS) attacks, spam, etc. [28, 48, 65, 66, 69, 79, 89, 104].

The TCP-SYN flood detector identifies attacks by monitoring the difference between the number of SYNs and FIN/RSTs received for a destination IP address within a certain time interval; high difference indicates the possibility of an attack [69, 104]. The port-scan detector detects the *possibility* of worm payload in both TCP and UDP packets by observing for port scan attacks [66, 69]; such suspicious packets are further analyzed to identify and create new worm signatures [66]. The worm signature matcher searches incoming packets for a given set of signatures using Wu and Manber's string matching algorithm [18, 108], and drops packets con-

taining a signature. We also implemented a load balancer that performs flow-level load balancing (for various flow definitions) across multiple IXP2400s to scale the throughput supported by our virtualized router.

We use the IXIA packet generator [9] for generating workload for the experiments in this chapter. Our IXIA generator contains two 1 gigabit ports that can generate a maximum of 2Gbps for our experiments.

5.2 Programmability and Performance

Table 5.1 shows the source lines of Baker code (SLOC) for each service. Each service took about a week to develop; most of the time was spent in understanding the behavior of the service, only a little time was spent in writing the Baker code. In contrast, developing a simple packet scheduler in microC on the IXP took us more than a month in the past (prior to Shangri-La). These observations demonstrate the power of Shangri-La in simplifying the development of high-performance packet processing services on programmable hardware. We attribute the reduction in development time to the high-level abstractions provided by ShaRE and, as a result, the simplicity of writing packet processing functions in Baker.

Table 5.1 also shows the throughput supported by one microengine and one IXP2400 (using four microengines) for each service—in kilo packets per second (Kpps) and mega bits per second (Mbps). In our current setup, two microengines are always used for the receive and transmit drivers that read-from/write-to the three gigabit ports, and two microengines are used for input packet classification and Ethernet encapsulation module. Hence, four microengines are available for other services. The table shows that the throughput scales linearly with the number of microengines; ShaRE maximizes the use of local resources to achieve high throughput.

Service	Baker SLOC	Throughput			
		1 ME		1 IXP	
		in Kpps	in Mbps	in Kpps	in Mbps
Load Balancer (64 byte pkts)	522	508	260	2030	1040
Port Scan Detector (64 byte pkts)	772	337	172	1273	652
TCP-SYN Flood Detector (64 byte pkts)	757	359	184	1385	709
Signature matcher (with worm, 1.5KB pkts)	615	17.5	210	68	816
Signature matcher (Amazon's index.html, 1.5KB pkts)	615	20	240	77	924

Table 5.1: Service benchmarks showing the source lines of code and the throughput achieved by different services.

5.3 Run-time Adaptation

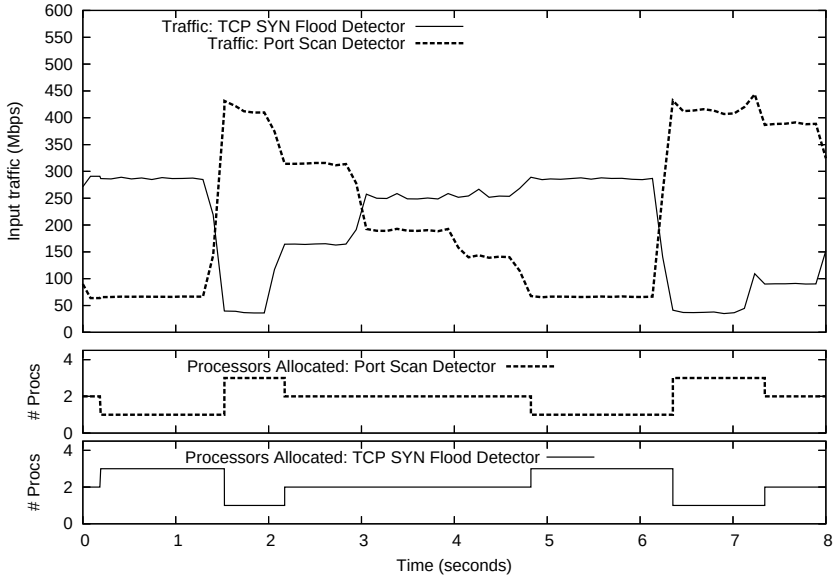


Figure 5.1: Experiment demonstrating adaptation of processors among services.

To demonstrate that ShaRE multiplexes processors effectively across competing services, we consider a deployment scenario with two services—port-scan detector and TCP-SYN flood detector. Each service is compiled as a single aggregate; the two aggregate share four microengines on an IXP2400. Among the other four microengines, two of them are reserved for the receive and transmit drivers, and two are used for the input classifier classifies packets as belonging either to the port-scan detector or to the TCP-SYN flood detector. We use IXIA packet generator [9] to simulate workload fluctuations for each aggregate (see the top graph in Figure 5.1). Figure 5.1 shows that as traffic mix changes with time, ShaRE adapts the processor allocation to match the demands of the two aggregates. During overload (between 2.2 and 4 seconds), processors are allocated to services according to proportional share; for this experiment, we chose the share to be equal for both services.

Chapter 6

Conclusion

6.1 Summary

In this dissertation, we describe the design and implementation of ShaRE, the first run-time system for such *high-performance, multi-service virtualized routers*. ShaRE advances the state-of-the-art in designing multi-service routers in two significant ways. First, ShaRE simplifies the programming of complex, multi-threaded multi-processor architectures, and thereby facilitates rapid development and deployment of high-performance network services. Second, ShaRE multiplexes available processing resources among competing services efficiently; this makes the system robust against traffic fluctuations and attacks, and reduces the overall processor provisioning requirements.

Although the requirements of multiplexing resources efficiently and improving programmability/portability are traditional operating system tasks, the domain of virtualized routers introduces several challenges that render conventional solutions to these problems ineffective. First, network services are required to process packets within a small delay from their arrivals; the delay tolerance is generally of the order of hundreds of microseconds to a few milliseconds. Second, at these timescales, net-

work traffic is bursty and often hard-to-predict. Further, the arrival rate of packets at virtualized routers is at least two to three orders of magnitude greater than in conventional environments. Third, the context-switch overhead of reallocating processing resources from one service to another in current hardware is large and is often two to three orders of magnitude greater than the time required to process a packet. Finally, to support high performance networks, programmable router hardware (unlike conventional systems) includes a complex array of resources such as multiple many-core processors, multiple hardware threads per core, fixed-function coprocessors, complex memory hierarchies and multiple inter-processor communication mechanisms. Further, the programmable router architectures evolve with time to incorporate advanced features in hardware.

In designing ShaRE, we make three contributions. First, we design Everest, an *agile* and *wary* scheduler for allocating processing resources to network services in high performance routers built on state-of-the-art programmable hardware. Everest's agility allows us to cope with stringent delay tolerance and difficult-to-predict and significant fluctuations in packet arrival rates at small time-scales. Everest's wariness copes with the relatively large context switch overheads; it does not context switch a processor unless necessary. Consequently, Everest is able to outperform conventional schedulers by orders of magnitude in minimizing the number of packets that exceed a given delay tolerance. Second, we perform a design space exploration of different schedulers under various system, service and workload characteristics. We demonstrate that the efficacy of a scheduler depends on the relationship between various characteristics, and design variants of Everest that perform the best under different characteristics. Finally, we identify and exploit several unique characteristics of the network services to minimize the context-switch overhead of replicating services or check-pointing and migrating services from one processor to another in a real system built using state-of-the-art hardware.

6.2 Future Work

6.2.1 Extensions to ShaRE

ShaRE is a first and crucial step towards creating a programming environment that will make future generations of virtualized routers as easily programmable as today's workstations and servers; several extensions will help us realize the goal completely.

Light-weight Memory Management ShaRE currently only manages the allocation of *processing resources* across competing services. However, since network services can allocate, access and release memory (e.g., for flow-specific data) dynamically, providing isolation/protection in the presence of limited memory resources requires careful memory management. While several operating systems have addressed this problem in the past, a memory manager for virtualized routers must address the unique challenges raised by the domain. For instance, since the amount of time to process a packet is small, and is typically dominated by memory accesses, any overhead in providing protection for memory accesses can increase the packet processing time t_{pkt} significantly. Similarly, since packets for a service can arrive at high rates, allocation and release of memory resources should be efficient to minimize the impact of memory allocation on the throughput of the service. Hence, the design of a light-weight memory manager is crucial.

Incorporating Distributed and Heterogeneous Resources To build realistic high-performance virtualized routers, ShaRE requires extensions in two fronts.

- ShaRE currently assumes a shared-memory model for per-flow data and packets; i.e. all processors can equally access the per-flow data that is kept in the global shared memory, and all *instances* of an aggregate (running on different processors) share the same queue of packets. While this assumption is valid for the ENP2611 platform we use for our prototype, a real virtualized router

that supports high bandwidths and advanced network services can be more complex containing multiple processors on arrays of boards, with distributed memories. For instance, a router can contain several ENP2611 boards each containing an IXP2400 processor, or a next-generation board containing many IXP2400s, or a mixture of ENP2611 boards and next-generation boards, etc. In such configurations, since aggregates of services can be replicated across processors and boards, ShaRE should be generalized to consider a distributed memory model.

We observe that incorporating a distributed memory model does not require any changes to Everest; the scheduler only uses pRAL interfaces and methods to monitor queue lengths, arrival rate, etc. This observation reiterates the power of pRAL that separates platform-independent tasks of ShaRE from platform-dependent tasks. However, the distributed memory model requires additional mechanisms for maintaining distributed communication channels per aggregate, and yet expose a single channel interface. Maintaining distributed channels requires us to develop additional mechanisms and policies for load distribution across the channels, to maintain ordering among packets of each flow, and to migrate data across distributed memories. For instance, one solution to maintain order among packets of a flow is to *pin* flows to a processing core—i.e. force all packets of a flow to be processed on the same processing core. One solution to support data migration transparently is to implement a light-weight distributed shared memory [25, 73].

- As virtualized router hardware gets incrementally upgraded to support the ever-increasing bandwidth demands, and as the hardware features and capabilities evolve with time, virtualized routers will contain hardware with diverse resource capabilities (e.g., varying processing capacity, different type of processors and fixed-function accelerators). In order to support such heterogene-

ity, both Everest and the mechanisms have to be enhanced.

Extending the mechanisms and pRAL to include newer hardware features is dependent on the hardware itself, and hence has to be done as and when the hardware evolves. Extending Everest to incorporate heterogeneous processing capacities can be done by appropriately scaling the throughput provided by each processor when deriving Equations 3.3 and 3.7. Further, a viable policy for beneficiary and victim selection would prefer processing cores with smaller capacities for context switching; the amount of processing capacity wasted due to context switch will be lower.

In our ongoing work, we are exploring both mechanisms and policies to incorporate both the distributed memory model and heterogeneity.

6.2.2 Towards Commercial-scale Overlay-hosting Facilities

When combined with overlay networks and programmable router hardware, ShaRE provides the power to create commercial-scale overlay hosting facilities, just like today's conventional web hosting facilities, that will greatly lower the barrier for introducing new network architectures and services into the Internet.

Commercial-scale hosting facilities will allow deploying richer network services that can be incrementally and selectively adopted by users; overlay services can be accessed by users through proxies on a per-service basis, thereby enabling the power to opt-in and opt-out of services at fine-granularities. Such a facility (termed *overlay hosting substrate*) will contain a network of high-performance virtualized routers(Refer Figure 1.1). While Planetlab [17, 31, 82, 83] is a *prototype* of such a substrate, it currently operates at low levels of performance than desirable for commercial-scale hosting. To cater to real user traffic, it is necessary to build the substrate using programmable multi-core multi-threaded hardware.

In this direction, ShaRE takes the first step by providing OS support for mul-

tiplexing processing resources and providing isolation across services, and simplifying programmability and improving portability on multi-core hardware. A logical next step is to build a testbed that emulates a commercial-scale overlay hosting facility. The testbed should enable us to simultaneously install, test and deploy (for real use) new overlay services, while transparently managing processing resources, memory and link capacities. Observe that, several network testbeds have been built during the past decade [2, 3, 4, 6, 97]. However, none of them have been adequate for simultaneously testing and deploying novel services. *Production* testbeds, such as Internet2 [2], support large-volume traffic from real users, but are not suitable for experimenting with new network services because users of these testbeds expect good performance and reliability. *Research* testbeds like DETER [6], and Emulab [3], allow a controlled environment to experiment with novel network services under diverse set of network topologies, but they do not receive real user traffic, and hence the results are less indicative of real operational viability. In contrast, a commercial-scale overlay hosting testbed provides the flexibility of choosing a richer service if possible, and falling back to the default service if the richer service does not work; such a facility will enable testing and deploying services simultaneously.

Appendix A

Proactive Everest

Equation 3.6 indicates that if $D < t_{pkt} + t_{sw}$ the packet that triggers adaptation can *not* be serviced before its delay tolerance when using Everest. Further, the derivation of Q^j_{lim} in Equation 3.4 assumes that at the instant the queue length crosses the threshold, j processors are actively processing packets. However, due to non-zero context switch overhead, a subset of the j processors can still be switching context, and not actively processing packets.

We address the above two conditions by deriving a proactive variant of Everest (named Everest-Pro). Then, we demonstrate that Everest-Pro performs poorly compared to Everest; Everest is better in minimizing the packets that exceed their delay tolerance even with the above limitations.

A.1 Everest-Pro Algorithm

As determined by Equation 3.4, let Q^j_{lim} denote the maximum queue length that j processors can process within the maximum permitted packet-processing delay D.

$$Q^j_{lim} = j * \left(\left\lfloor \frac{D}{t_{pkt}} \right\rfloor - 1 \right) \tag{A.1}$$

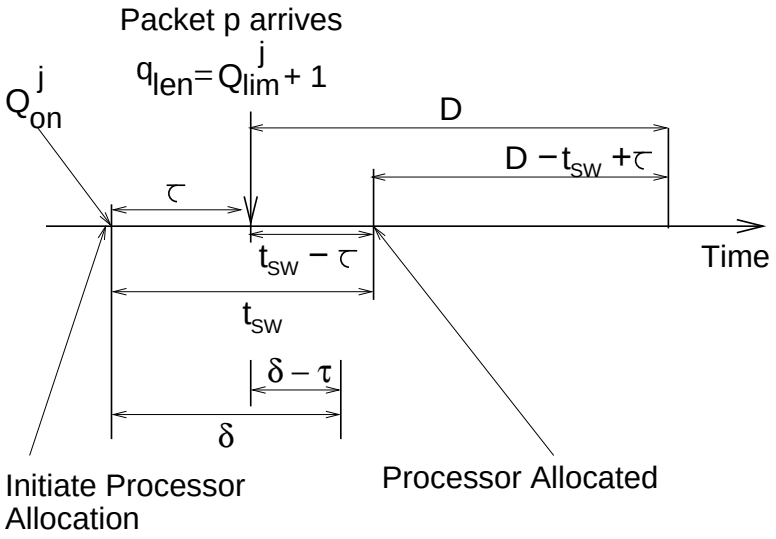


Figure A.1: Allocation procedure for Everest-Pro: Timing diagram

Let p be the packet that makes the queue length cross Q^j_{lim}. Unless a new processor is allocated, p will exceed its delay tolerance. To address the conditions outlined above, consider that the scheduler requests allocation of the new processor τ units of time *prior* to the arrival of the packet p (see Figure A.1). When $\tau > 0$, Everest *speculatively* allocates additional processors in anticipation of q_{len} exceeding Q^j_{lim}; in contrast, when $\tau \leq 0$ Everest allocates additional processors only after observing that $q_{len} > Q^j_{lim}$. Let the new threshold be called Q^j_{th}.

$$\begin{aligned} Q^j_{th} &= Q^j_{lim} \quad if\, \tau \geq 0 \\ &= Q^j_{lim} - (\tau \times R_{arr}(\Delta) - P^j_{dep}(\tau)) \quad otherwise \end{aligned} \tag{A.2}$$

Also, let us assume that there is one processor that was previously allocated that is still switching its context to the current aggregate, i.e. only $j-1$ processors are actively processing packets. Let δ be the amount of time left before the j^{th} processor is active to process packets at the instant when the queue reaches Q^j_{th}. Then, at the instant p arrives, the j^{th} processor will be ready to process packets after $\delta - \tau$ time units.

From Figure A.1, for packet p to be processed within D, the number of packets processed by the currently allocated processors (including the context-switching processors) and the new processor in the next D units of time should include the packet p.

- The number of packets that the j processors can process can be written as

$$P_{dep}^{j}(D) - \left\lceil \frac{\delta - \tau}{t_{pkt}} \right\rceil \tag{A.3}$$

  where the second term represents the number of packets that could not be processed by the j^{th} processor because of context switching. The above expression can be generalized to k previously allocated processors that are still context-switching when packet p arrives as

$$P_{dep}^{j}(D) - \sum_{y=1}^{k} \left\lceil \frac{\delta_y - \tau}{t_{pkt}} \right\rceil \tag{A.4}$$

  where δ_y is the amount of time left for the y^{th} processor to start processing packets when the queue reaches Q_{th}^{j}.

- The number of packets that the $(j+1)^{th}$ processor will process after context switching is equal to

$$P_{dep}^{1}(D - t_{sw} + \tau) \quad = \quad \left\lfloor \frac{D - t_{sw} + \tau}{t_{pkt}} \right\rfloor \tag{A.5}$$

Using Expressions A.4 and A.5, the condition for packet p to be processed within its delay tolerance is

$$Q_{lim}^{j} + 1 + j \quad \leq \quad P_{dep}^{j}(D) - \sum_{y=1}^{k} \left\lceil \frac{\delta_y - \tau}{t_{pkt}} \right\rceil + \left\lfloor \frac{D - t_{sw} + \tau}{t_{pkt}} \right\rfloor \tag{A.6}$$

Since $-\lceil A \rceil \leq -\lfloor A \rfloor$ for any real number A, Equation A.6 can be rewritten as

$$Q_{lim}^{j} + 1 + j \quad \leq \quad P_{dep}^{j}(D) - \sum_{y=1}^{k} \left\lfloor \frac{\delta_y - \tau}{t_{pkt}} \right\rfloor + \left\lfloor \frac{D - t_{sw} + \tau}{t_{pkt}} \right\rfloor \tag{A.7}$$

Since $\lfloor A \rfloor - \lfloor B \rfloor \leq \lfloor A - B \rfloor$ for any real numbers A and B, Equation A.7 can be rewritten as

$$Q^j_{lim} + 1 + j \quad \leq \quad P^j_{dep}(D) + \left\lfloor \frac{D - t_{sw} + (k+1) \times \tau - \sum_{y=1}^{k} \delta_y}{t_{pkt}} \right\rfloor \tag{A.8}$$

Substituting Equation A.1 in Equation A.8 and simplifying, we get

$$1 \quad \leq \quad \left\lfloor \frac{D - t_{sw} + (k+1) \times \tau - \sum_{y=1}^{k} \delta_y}{t_{pkt}} \right\rfloor \tag{A.9}$$

$$\Rightarrow \tau \quad \geq \quad \frac{t_{pkt} + t_{sw} - D + \sum_{y=1}^{k} \delta_y}{k+1} \tag{A.10}$$

A.2 Comparison

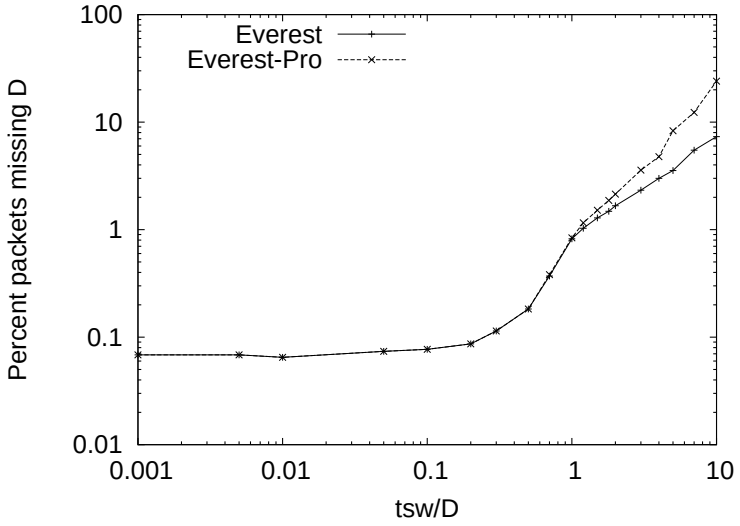


Figure A.2: Graph demonstrating that greater number of packets will exceed their delay tolerance with a proactive scheduler.

Figure 3.8 compares the percentage of packets that suffer delays greater than the delay tolerance with varying ratio of t_{sw} and D when using Everest and Everest-Pro. The system contains 16 processors, D is set to 1ms, and the workload used is UNC-2.5 trace, and t_{sw} is varied. The graph shows that when $D > t_{pkt} + t_{sw}$, i.e.

when $\frac{t_{sw}}{D} < 1$, Everest-Pro performs identical to Everest. Both Everestand Everest-Pro allocate a new processor for an aggregate only when $\tau < 0$, i.e. both are reactive. When $D < t_{pkt} + t_{sw}$, i.e. when $\frac{t_{sw}}{D} > 1$, Everest-Pro (by being proactive) causes more packets to exceed their delay tolerance than Everest. The difference between Everest and Everest-Pro increases with t_{sw}.

To understand the above observation better, Figure A.3 plots the cumulative distribution of the amounts of time the request for a processor is denied (because it is allocated to another aggregate or is context-switching) before being satisfied. As demonstrated by the graph, the request denial time increases substantially with increasing t_{sw} when using Everest-Pro. We made similar observations for different values of D and with different traces. These observations demonstrate that Everest-Pro (a more proactive scheduler) does not provide any additional benefits over Everest, despite its complexity (Section A.1), thereby substantiating the design choices we made when developing Everest.

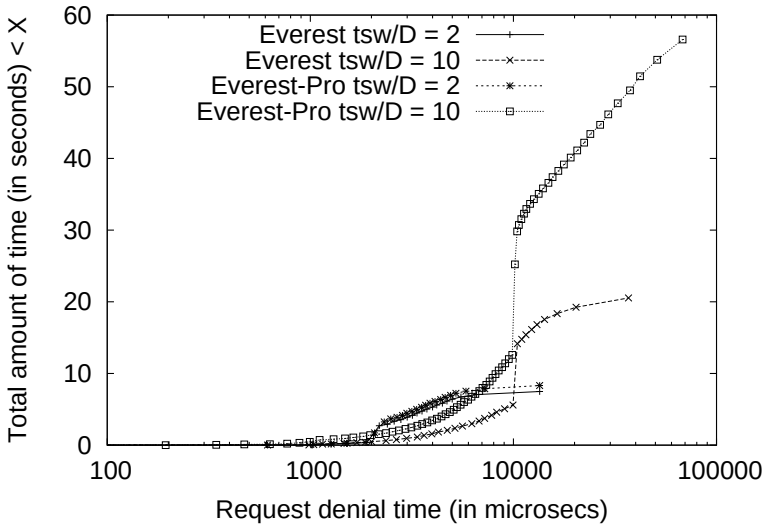


Figure A.3: Denial CDF demonstrating that being proactive increases the time for which processors are denied before being granted.

Bibliography

[1] Akamai Technologies, Inc. http://www.akamai.com/.

[2] The Abilene Network. http://abilene.internet2.edu/.

[3] The Emulab Testbed. http://www.emulab.net.

[4] Active Network Backbone (ABone). http://www.isi.edu/abone/.

[5] CloudShield Technologies. http://www.cloudshield.com.

[6] DETER. http://www.isi.edu/deter/.

[7] Intel IXA Software Developers Kit 3.0.
http://www.intel.com/design/network/products/npfamily/sdk3.htm.

[8] Introduction to the Auto-Partitioning Programming Model: White paper.
http://www.intel.com/design/network/papers/254114.htm.

[9] IXIA. http://www.ixiacom.com.

[10] Linux-based user-space NAT-PT.
http://www.ipv6.or.kr/english/natpt-overview.htm.

[11] MontaVista Software. http://www.mvista.com.

[12] NEC Develops Programmable Parallel Processor Chip for In-Vehicle Video Image Recognition Applications.
http://www.nec.co.jp/press/en/0302/1001.html.

[13] Network Virtualization: a strategy for de-ossifying the internet .
http://www.arl.wustl.edu/netv/.

[14] Payloadplus family of network processors. http://www.agere.com/enterprise_metro_access/network_processors.html.

[15] Platform 2015: Intel Processor and Platform Evolution for the Next Decade.
ftp://download.intel.com/technology/computing/archinnov/platform2015/download/Platform_2015.pdf.

[16] RadiSys ENP2611 data sheet.
http://www.radisys.com/oem_products/ds-page.cfm?productdatasheetsid=1147.

[17] Report of NSF workshop on Overcoming Barriers to Disruptive Innovation in Networking, January 2005.
http://www.arl.wustl.edu/netv/noBarriers_final_report.pdf.

[18] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org/.

[19] STI Cell Processor.
http://www-1.ibm.com/businesscenter/venturedevelopment/us/en/featurearticle/gcl_xmlid/8649/nav_id/emerging.

[20] Sun's Multi-Core Plans.
http://www.aceshardware.com/read_news.jsp?id=60000498.

[21] TejaNPTM: A Software Platform for Network Processors. http://www.teja.com.

[22] The Definitive ISP buyer's guide. `http://www.thelist.com/`.

[23] Tarek F. Abdelzaher, Kang G. Shin, and Nina Bhatti. Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[24] M. Adiletta, D. Hooper, and M. Wilde. Packet Over SONET: Achieving 10 Gigabit/sec Packet Processing with IXP2800. *Intel Technology Journal*, 6(3), 2002.

[25] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

[26] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. *SIGOPS Oper. Syst. Rev.*, 35(5):131–145, 2001.

[27] J. H. Anderson and A. Srinivasan. Pfair scheduling: beyond periodic task systems. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications (RTCSA'00)*, page 297, Washington, DC, USA, 2000. IEEE Computer Society.

[28] Tom Anderson, Timothy Roscoe, and David Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[29] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Server. In

Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems, 2000.

[30] C. Barakat, P. Thiran, G. Iannaccone, C. Diot, and P. Owezarski. Modeling internet backbone traffic at the flow level. *IEEE Transactions on Signal processing. Special Issue in Networking*, 51(8), 2003.

[31] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *Symposium on Networked Systems Design and Implementation, San Francisco CA, Mar 2004.*

[32] Jon C. R. Bennett and Hui Zhang. Hierarchical packet fair queueing algorithms. *IEEE/ACM Trans. Netw.*, 5(5):675–689, 1997.

[33] Marjory S. Blumenthal and David D. Clark. Rethinking the design of the internet: the end-to-end arguments vs. the brave new world. *ACM Trans. Inter. Tech.*, 1(1):70–109, 2001.

[34] Vijay Bollapragada, Russ White, and Curtis Murphy. *Inside Cisco IOS Software Architecture*, chapter 1. Cisco Press, 2000.

[35] A. T. Campbell, S. Chou, M. E. Kounavis, V. D. Stachtos, and J. Vicente. NetBind: A Binding Tool for Constructing Data Paths in Network Processor-Based Routers. In *Proceedings of the Fifth Interational Conference on Open Architectures and Network Programming (OPENARCH' 02)*, June 2002.

[36] A. T. Campbell, H. De Meer, M. E. Kounavis, K. Miki, J. Vicente, and D. Villela. The genesis kernel: A virtual network operating system for spawning network architectures. In *2nd IEEE International Conference on Open Architectures and Network Programming*, 1999.

[37] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A proportional-share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI*, 2000.

[38] A. Chandra, M. Adler, and P. Shenoy. Deadline fair scheduling: Bridging the Theory and Practice of Proportionate Fair Scheduling in Multiprocessor Systems. In *IEEE Real Time Technology and Applications Symposium*, 2001.

[39] A. Chandra, W. Gong, and P. Shenoy. Dynamic Resource Allocation for Shared Data Centers Using Online Measurement. In *Int. Workshop on Quality of Service*, 2003.

[40] Jeffrey S. Chase, Darrell Anderson, Prachi Thakar, Amin Vahdat, and Ronald Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[41] Hemant M. Chaskar and Upamanyu Madhow. Fair scheduling with tunable latency: a round-robin approach. *IEEE/ACM Trans. Netw.*, 11(4):592–601, 2003.

[42] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-la: achieving high performance from compiled network applications while enabling ease of programming. *SIGPLAN Not.*, 40(6):224–236, 2005.

[43] Su-Hui Chiang and Mary K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200–223. Springer-Verlag, 1996.

[44] Guo Chuanxiong. Srr: An o(1) time complexity packet scheduler for flows in multi-service packet networks. In *SIGCOMM '01: Proceedings of the 2001*

conference on Applications, technologies, architectures, and protocols for computer communications, pages 211–222, New York, NY, USA, 2001. ACM Press.

[45] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole. Fast concurrent dynamic linking for an adaptive operating system. In *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 108. IEEE Computer Society, 1996.

[46] Dan Decasper, Zubin Dittia, Guru M. Parulkar, and Bernhard Plattner. Router Plugins: A Software Architecture for Next Generation Routers. In *Proceedings of ACM SIGCOMM*, 1998.

[47] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[48] Cristian Estan and George Varghese. New Directions in Traffic Measurement and Accounting. In *Proceedings of ACM SIGCOMM*, Pittsburgh, PA, August 2002.

[49] Michael J. Freedman, Eric Freudenthal, and David Mazières. Democratizing content publication with coral. In *NSDI*, pages 239–252, 2004.

[50] Lal George and Matthias Blume. Taming the ixp network processor. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 26–37. ACM Press, 2003.

[51] S. Jamaloddin Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. IEEE INFOCOM'94*, volume 2, pages 636 – 646, 1994.

[52] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Syst.*, 25(2-3):187–205, 2003.

[53] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *SIGOPS Oper. Syst. Rev.*, 33(5), 1999.

[54] P. Goyal, X. Guo, and H.M. Vin. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *USENIX Symposium on Operating Systems Design and Implementation*, 1996.

[55] Pawan Goyal, Harrick M. Vin, and Haichen Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *SIGCOMM '96: Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 157–168, New York, NY, USA, 1996. ACM Press.

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

[57] Intel IXP Family of Network Processors. http://www.intel.com/design/network/products/npfamily/index.htm.

[58] Sitaram Iyer and Peter Druschel. Anticipatory scheduling: a disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 117–130, New York, NY, USA, 2001. ACM Press.

[59] Sugih Jamin, Peter B. Danzig, Scott Shenker, and Lixia Zhang. A measurement-based admission control algorithm for integrated services packet networks. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 2–13. ACM Press, 1995.

[60] Wei Jin, Jeffrey S. Chase, and Jasleen Kaur. Interposed proportional sharing for a storage service utility. In *Joint international conference on Measurement and modeling of computer systems*, 2004.

[61] Erik J. Johnson and Aaron Kunze. *IXP2400/2800 Programming*. Intel Press, 2003.

[62] V. Kanodia and E. Knightly. Multi-class latency-bounded web services. In *In Proceedings of the 8th International Workshop on Quality of Service*, 2000.

[63] Scott Karlin and Larry Peterson. VERA: an extensible router architecture. *Computer Networks*, 38(3), 2002.

[64] Ralph Keller, Lukas Ruf, Amir Guindehi, and Bernhard Plattner. PromethOS: A Dynamically Extensible Router Architecture Supporting Explicit Routing. In *Proceedings of Fourth Annual International Working Conference on Active Networks (IWAN)*, 2002.

[65] Angelos D. Keromytis, Vishal Misra, and Dan Rubenstein. Sos: secure overlay services. *SIGCOMM Comput. Commun. Rev.*, 32(4):61–72, 2002.

[66] H-A. Kim and B. Karp. Autograph: Toward automated, distributed worm signature detection. In *Proc. USENIX Security '04*, pages 271–286, 2004.

[67] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[68] Ravi Kokku, Taylor Riche, Aaron Kunze, Jayaram Mudigonda, Jamie Jason, and Harrick Vin. A case for run-time adaptation in packet processing systems. *ACM SIGCOMM Computer Communication Review*, 34(1):107–112, January 2004.

[69] R.R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In *Proc. ACM Internet Measurement Conference (IMC) '04*, pages 187–200, 2004.

[70] Michael E. Kounavis, Andrew T. Campbell, Stephen T. Chou, and John Vicente. Programming the Data Path in Network Processor-Based Routers. *Software Practice and Experience*, 2004.

[71] J. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *USENIX Annual Technical Conference*, 2002.

[72] K. Li, J. F. Naughton, and J. S. Plank. Real-time, concurrent checkpoint for parallel programs. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 79–88, 1990.

[73] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Trans. Comput. Syst.*, 7(4):321–359, 1989.

[74] Giuseppe Lipari and Sanjoy K. Baruah. Efficient scheduling of real-time multitask applications in dynamic systems. In *RTAS '00: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 166, Washington, DC, USA, 2000. IEEE Computer Society.

[75] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[76] Chenyang Lu, Guillermo A. Alvarez, and John Wilkes. Aqueduct: Online data migration with performance guarantees. In *FAST '02: Proceedings of the Conference on File and Storage Technologies*, pages 219–230, Berkeley, CA, USA, 2002. USENIX Association.

[77] D. Mosberger. *Scout: A Path-Based Operating System.* PhD thesis, The University of Arizona, July 1997.

[78] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[79] R. Pang, V. Yagneshwaran, P. Barford, V. Paxon, and L. Peterson. Characteristics of Internet Background Radiation. In *ACM Internet Measurement Conference*, 2004.

[80] Przemyslaw Pardyak and Brian N. Bershad. Dynamic binding for an extensible system. In *Proceedings of the second USENIX symposium on Operating systems design and implementation*, 1996.

[81] KyoungSoo Park, Vivek S. Pai, Larry L. Peterson, and Zhe Wang. Codns: Improving dns performance and reliability via cooperative lookups. In *OSDI*, pages 199–214, 2004.

[82] L. Peterson, D. Culler, T. Anderson, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *In Proceedings of the 1st Workshop on Hot Topics in Networks (HotNets-I)*, 2002.

[83] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. In *HOTNETS III*, 2004.

[84] Yi Qiao, Jason Skicewicz, and Peter Dinda. Multiscale Predictability of Network Traffic. Northwestern University. Technical report.

[85] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[86] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[87] Niraj Shah, William Plishker, and Kurt Keutzer. NP-Click: A Programming Model for the Intel IXP1200. In *Proceedings of the 2nd Workshop on Network Processors (NP2)*, February 2003.

[88] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round Robin. In *Proceedings of ACM SIGCOMM*, August 1995.

[89] S. Singh, C. Estan, G. Varghese, and S. Stefage. Automated worm fingerprinting. In *Proc. OSDI '04*, pages 45–60, 2004.

[90] F. Donelson Smith, Félix Hernández Campos, Kevin Jeffay, and David Ott. What TCP/IP Protocol Headers Can Tell Us About the Web. In *Proceedings of ACM SIGMETRICS 2001/Performance 2001*, June 2001.

[91] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[92] Anand Srinivasan, Philip Holman, James Anderson, Sanjoy Baruah, and Jasleen Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *2nd Workshop on Network Processors*, 2003.

[93] Dimitrios Stiliadis and Anujan Varma. Efficient fair queueing algorithms for packet-switched networks. *IEEE/ACM Trans. Netw.*, 6(2):175–185, 1998.

[94] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *RTSS '96: Proceedings of the 17th IEEE Real-Time*

Systems Symposium (RTSS '96), page 288, Washington, DC, USA, 1996. IEEE Computer Society.

[95] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. In *ACM SIGCOMM*, 2002.

[96] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy H. Katz. Overqos: offering internet qos using overlays. *SIGCOMM Comput. Commun. Rev.*, 33(1):11–16, 2003.

[97] Joe Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Computer Networks*, pages 117–135, July 2001.

[98] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). IETF RFC 2766, February 2000.

[99] P. Unnikrishnan, G. Chen, M. Kandemir, and D. R. Mudgett. Dynamic compilation for energy adaptation. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 158–163, New York, NY, USA, 2002. ACM Press.

[100] Bhuvan Urgaonkar, Prashant J. Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation*, 2002.

[101] D. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. In *Proc. TRICOMM'91*, 1991.

[102] H. Vin, J. Mudigonda, J. Jason, E. J. Johnson, R. Ju, A. Kunze, and R. Lian. A programming environment for packet-processing systems: Design considerations. In *3rd Workshop on Network Processors and Applications*, February 2004.

[103] C. A. Waldspurger. Lottery and stride scheduling: Flexible proportional-share resource management. Technical Report MIT/LCS/TR-667, 1995.

[104] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proc. INFOCOM '02*, pages 1530–1539, 2002.

[105] L. Wang, V. Pai, and L. Peterson. The Effectiveness of Request Redirection on CDN Robustness. In *Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[106] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP)*, October 2001.

[107] Richard West and Karsten Schwan. Dynamic window-constrained scheduling for multimedia applications. In *IEEE International Conference on Multimedia Computing and Systems*, 1999.

[108] Sun Wu and Udi Manber. Fast text searching: allowing errors. *Commun. ACM*, 35(10):83–91, 1992.

[109] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *ACM SIGCOMM*, Aug 2004.

[110] Y. Yemini and S. daSilva. Towards programmable networks. In *IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, 1996.

[111] Z. Zhang, V. Ribeiro, S. Moon, and C. Diot. Small-Time Scaling Behaviors of Internet Backbone Traffic: An Empirical Study. In *Proceedings of the IEEE INFOCOM.*, 2003.

Vita

Ravi Kokku was born in Hyderabad, India, on August 1, 1976, of Lakshmi Narayana and Santha Kumari. He studied at the Atomic Energy Central School and Junior College in Hyderabad for the first 12 academic years. He then attended the Indian Institute of Technology at Kharagpur, where he received the Bachelor of Technology in Computer Science and Engineering in June 1997. Thereafter, he was a software engineer and senior software engineer for two and a half years at Hughes Software Systems, contributing to the design and development of satellite communication systems.

In January 2000, he moved to the University of Texas at Austin to pursue his graduate studies. He was awarded the IBM doctoral fellowship from Fall 2001 to Spring 2004. He received Master of Science in Computer Science in May 2004, and Doctor of Philosophy in Computer Science in August 2005. He is all set to pursue yet another career in his life as a researcher at NEC labs, Princeton, NJ, starting September 2005.

Permanent Address: c/o K. Lakshmi Narayana H.No 1-7-77
Kamalanagar, E.C.I.L. (po)
Hyderabad 500062
Andhra Pradesh, INDIA

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.