1-1-2011

High Speed Networking In The Multi-Core Era

Benjamin Wun
Washington University in St. Louis

J

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Patrick Crowley, Chair
James Buckley
Roger Chamberlain
David Deal
Chris Gill
Viktor Gruev

HIGH SPEED NETWORKING IN THE MULTI-CORE ERA

by

Benjamin Wun

A dissertation presented to the Graduate School of Arts and Sciences
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2011
Saint Louis, Missouri

ABSTRACT OF THE THESIS

High Speed Networking in the Multi-Core Era

by

Benjamin Wun

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2011

Research Advisor: Professor Patrick Crowley

High speed networking is a demanding task that has traditionally been performed in dedicated, purpose built hardware or specialized network processors. These platforms sacrifice flexibility or programmability in favor of performance. Recently, there has been much interest in using multi-core general purpose processors for this task, which have the advantage of being easily programmable and upgradeable. The best way to exploit these new architectures for networking is an open question that has been the subject of much recent research. In this dissertation, I explore the best way to exploit multi-core general purpose processors for packet processing applications. This includes both new architectural organizations for the processors as well as changes to the systems software. I intend to demonstrate the efficacy of these techniques by using them to build an open and extensible network security and monitoring platform that can out perform existing solutions.

Acknowledgments

I would like to start by thanking my advisor Patrick Crowley for all the advice and support he has given me over the years. His perpetual optimism and practical knowledge have been invaluable.

I have had the opportunity to learn from many of the excellent faculty in this department, including Roger Chamberlain, Jeremy Buhler, and John Turner. I am grateful for the knowledge they and many others imparted to me. I would also like to thank the members of my committee, Chris Gill, Viktor Gruev, David Deal and Jim Buckley.

The CSE office staff have helped to make my life easier in so many ways. Thanks go to Madeline Hawkins, Jayme Moehle, Sharon Matlock, Myrna Harbison and Kelli Eckman.

I would also like to thank the ARL staff, especially John Dehart and Fred Kuhns, who have provided me with excellent advice and practical assistance. This work would have been impossible without their help.

My fellow students have been wonderful to work with, especially Brandon, Eric, Mart, Shakir, Haowei, Michael, Arpith, Michela, Charlie and Mike. I have greatly enjoyed bouncing ideas off of them and mining their practical skills for my own benefit. I hope they have gotten as much out of interacting with me as I have with them. A special thanks goes to Charlie for his assistance with ONL and Mike for collaborating with me on the Scheduler API.

I had the great fortune to spend a year interning at Intel in Oregon where I met and worked with many excellent researchers. I would like to thank them all for the opportunity and for all that they taught me, especially Annie, Arun and Erik, who had a tremendous impact on the work presented here.

Finally, my parents have been unwavering in their support of me and taught me to be curious about the world around me. For that, I am forever grateful.

Benjamin Wun

Washington University in Saint Louis
December 2011

Dedicated to my parents Lap-Ming and Mei-Na Wun.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

High speed packet processing is a demanding but important task that is usually performed in dedicated, purpose built hardware. As networks get more complex, additional tasks such as network security and monitoring, or the deployment of research protocols, have created a demand for programmable network devices that can be extended by the end user to perform new tasks or test new ideas. The existing solutions to this problem range from software routers based on general purpose processors, to specialized network processors and extensions to commercial routers. No single solution delivers the combination of flexibility, programmability and high performance required in this domain.

Recently, general purpose processors (GPPs) have been adopting characteristics common to network processing, especially the use of multi-threaded processing cores and multiple cores on a chip. These additions make conventional processors better able to meet the requirements of high speed networking applications; furthermore, general purpose processors are easier to program than network processors (NPs). GPPs can use standard operating systems and programming languages, and do not expose architectural details to the extent that NPs do. However, the software frameworks on GPPs are often not optimized for multi-core environments, or for the demanding and specialized task of line rate network processing. Many projects have tried to make it easier to write networking code on GPPs, but none seem to offer the definitive solution.

In this dissertation, I will explore new architectures, both software and hardware, for designing programmable high speed network processing platforms. I will further demonstrate the effectiveness of such an architecture for creating network monitoring applications.

The first part of this study will investigate architectural additions to general purpose processors to support networking. I propose and evaluate a heterogeneous multi-core architecture that moves network stack processing to a series of specialized cores on the same processor die as the main CPU.

The second part of this study will examine the best way to create programmable packet processing platforms in the context of existing general purpose platforms. I examine the efficacy of various parallelization strategies on Snort, an example of a complex and stateful network application. I present an API for writing complex, stateful, pipelined network applications and examine its efficacy by porting the Snort intrusion detection system to use it. Finally, I take the lessons learned from parallelizing Snort and apply them to the design of a scheduler API for network centric applications.

1.1 Contributions

This thesis makes several contributions.

First, we propose a novel hardware architecture that uses a cluster of specialized cores to perform networking tasks on behalf of a host processor, a technique known as network onloading. We evaluate this architecture by building a prototype using the IXP network processor as a starting point. We demonstrate an improvement in packet reception throughput from 40 to 100 % depending on the incoming packet size.

Second, we propose an API for writing networking applications on multi-core architectures. We evaluate the effectiveness of this API by porting the Snort intrusion detection system to use it.

Finally, we extend the API to include provisions for writing new schedulers that dynamically adapt the application mapping to the underlying hardware based on changes in the workload. We use this API to evaluate the effectiveness of two different scheduling algorithms and demonstrate the usefulness of each under different scenarios.

1.2 Methodology

We test the proposed ideas by building working prototypes. Evaluating the prototypes can happen in a variety of ways. When possible, systems were tested using the Open Network Laboratory, a reprogrammable network testbed [51]. Other systems were evaluated in a more ad-hoc fashion. The details of each experiment are described in further detail in the appropriate chapter.

1.3 Organization

This document is organized around a series of projects in which ideas are proposed and prototypes built to test them. Each chapter represents a project organized around a single idea, and as such, background information and related work for each project are presented in the relevant chapter. The remainder of this document is organized as follows. Chapter 2 proposes a novel architecture for network onloading. Chapter 3 presents an evaluation of a parallelized version of Snort and an analysis of the bottleneck encountered by it. Chapter 4 presents an API for writing parallelized network applications. Chapter 5 extends this API to allow the writing of new packet schedulers and evaluates the efficacy of two different schedulers, one of which has not previously been published.

Chapter 2

Hardware Acceleration

In this section, we examine the proposition that the addition of small, simple cores to a general purpose CPU can accelerate standard sockets network I/O, either by employing the techniques of server NICs or through network onloading, which moves network protocol processing from the host CPU (which must be shared with other applications) to a set of dedicated resources. The goal of this proposed architecture is to find a way to preserve traditional network programming semantics while bringing performance in line with modern demands. To evaluate this proposal, we have built a prototype system using the Intel IXP network processor [22]. We will begin this section by providing some background about the IXP, then provide some further background on network processing on general purpose processors, discuss related work, and finish by describing our proposed architecture.

2.1 Packet Processing and the Intel IXP

IXP Network Processors (NPs) feature two types of processors. The first is an ARM based XScale which boots a traditional OS and is typically used in management and slow path processing. The second processor type, the microengine (ME), is a small embedded core for line-rate packet processing. IXP NPs have a single XScale, and 4, 8, or 16 MEs, depending on the specific chip. In this work, we use the IXP2350, which is illustrated in Figure 2.1.

In our prototype, the XScale is the host CPU, and the MEs provide I/O acceleration. This is an atypical use of the XScale. In most application, the XScale would only

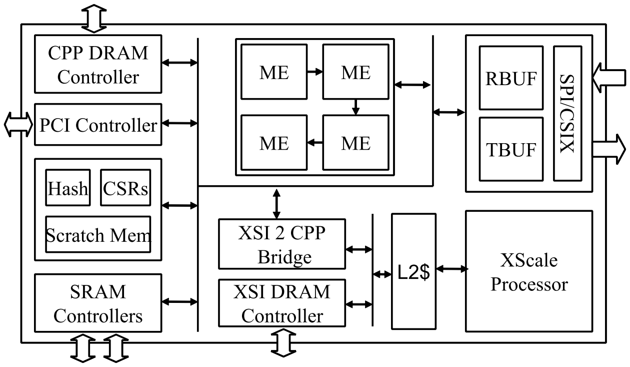


Figure 2.1: Organization of the IXP 2350 NP

receive exception packets, a relatively small fraction of traffic. In our system, an application on the XScale is the source and destination for every packet. We use the IXP for its convenient approximation of our architectural model. The XScale represents a high-performance GPP for which the MEs accelerate network I/O.

Each IXP ME provides hardware support for 8 hardware thread contexts, including register storage, multi-threading ISA extensions, and a thread arbiter. Each ME has its own local data and instruction storage, both implemented as SRAMs. An ME communicates asynchronously with other units via I/O commands and transfer registers. A DRAM read, for example, is carried out by sending a read operation to the DRAM controller (via the Command Outlet FIFO) that specifies the desired address as well as the target incoming transfer registers to which the data should be delivered. Hardware signals are specified in the ISA and are asserted when requested operations have completed. This message-passing style and the use of hardware signals allow ME software to initiate multiple external requests without blocking, as long as subsequent computation does not depend on the completion of these requests. This interface provides both a more efficient way to access memory and a way to hide memory latencies.

Other units provide critical functions or resources in hardware, including a configurable hash unit, 16KB of on-chip scratch memory and 128KB of message SRAM.

The IXP 2350 include a DDR SDRAM and a QDR SRAM controller on-chip as channels for bulk and latency-sensitive data storage, respectively. A separate channel of SDRAM is used by the OS and programs on the XScale. Both the MEs and XScale are clocked at 900 MHz.

All IXP processors contain a Media Switch Fabric (MSF) to facilitate high speed communication between the MEs and MACs. The IXP2350 uses the MSF to interface to its two, 1Gbps on-chip MACs. Having the MAC located on-chip over a high speed interface is a great advantage for scalable, high speed networking [10].

2.2 Linux Network Stack

On general purpose processors, network processing is split between the network interface card (NIC) and the network stack running in the operating system kernel on the host CPU. In this section we provide some brief background on how packets flow through a modern system.

First, we take a look at packet reception. When a packet arrives at the NIC, it raises an interrupt, which causes the host to stop whatever program it is executing in order to run the network receive code. The host copies the packet from the NIC to a buffer in main memory. From there, the headers and checksums are verified, the packet is classified and the payload is copied to the buffer of the user program that is waiting for it.

On the transmit side, the host copies packets from a program's buffer into a protected kernel space buffer. From there, it adds the proper headers and computes checksums. It then copies the packet into the NIC's buffer and tells it to transmit the packet.

With this data flow, all processing is done on the host processor, which cannot execute other programs while it is handling network traffic. Since network processing shares the CPU with other programs, under load, the system may drop a large number of packets or become so dominated by network processing that other programs are starved of resources. Modern NICs include accelerators to offload or streamline parts of this process, including interrupt moderation, receive rings and checksum offload.

These are discussed in more detail in our description of our prototype system in section 2.4. In network onloading, we take this a step further and move all network protocol processing to dedicated resources, freeing the host CPU to perform other tasks, as discussed in section 2.4.2.

A further bottleneck in this scenario is the movement of data from user to kernel space. Not only does it require moving a potentially large amount of data, it requires a context switch from kernel to user space and back, which can be very costly- up to 36 percent in some scenarios [23]. However, this is unavoidable if we are to preserve the sockets programming model, which is an explicit goal of this part of the project.

2.3 Related Work

TCP offload engines (TOEs), which move protocol processing from the host CPU to the NIC, are being used to accelerate specific tasks, such as storage area networking or for use with protocols such as RDMA [33] [48]. Though commercial implementations exist, it is inconclusive whether TOEs are actually an effective solution, with some studies showing the TOE itself to be the actual bottleneck [5] [43]. Our approach differs from that taken with TOEs, as IXP MEs are on-chip, fully-programmable, and closely coupled with the CPU, thus bypassing the major problems with TOEs and providing additional opportunities for optimization. Furthermore, our interest is in accelerating general purpose networking, whereas most TOEs are used to accelerate a specific task.

Binkert et al., in a simulation based study, have examined the efficacy of moving the NIC's location relative to the CPU [10]. They found that putting the NIC on a direct HyperTransport like channel and eliminating the I/O bus bottleneck greatly increased system throughput. Locating the NIC on chip produced further improvements for the receive path and also allowed packet data to be directly written into cache, a potential accelerator for certain network workloads. While the IXP has no mechanism for direct cache access by the MEs, the MEs, MSF, and MACs are located on-chip with a dedicated off-chip connection to the PHY.

The ETA [41] project demonstrates a new interface for communication between a host processor and an associated packet processing engine (PPE). Their interface allows for asynchronous operation, whereby a user program can send off a packet for transmission or request notification of arriving packets without blocking. The traditional sockets interface semantics require a program to block until the packet is sent or until a packet arrives. For the prototype ETA system, the PPE was a Xeon processor in an SMP system. This prototype showed both improved network throughput and an increase in surplus cycles for the host processor. This differs from our research because ETA uses a state of the art superscalar out of order processor with a high clock rate, deep pipeline and prodigious amounts of cache as the packet processing engine, whereas our project uses the smaller, in order, single issue MEs of the IXP. The smaller, simpler MEs are more power and area efficient for this task than the Xeon.

The authors of ETA have advocated TCP Onloading by combining ETA with a memory aware reference stack (MARS) [41]. MARS is an attempt to mitigate memory access latencies by using asynchronous memory copies, light-weight threading, and direct cache access. The first two are already present in the IXP.

Finally, recent advances in NIC architectures for virtualization have produced commercial NICs with multiple receive queues and the ability to classify packets at the NIC. This pushes some functionality traditionally performed in the CPU to the NIC and provides an alternative approach to exploiting multi-core processors. This approach is implemented in homogeneous multi-core environments (though it does not have to be) and packets are still processed in a traditional network stack (either in the OS or VM) and only yields real performance gains when there are a large number of flows involved which can be processed in parallel.

2.4 Network Acceleration

In order to evaluate the effectiveness of our onloading proposal, we have explored two ways of using the MEs to accelerate network processing in the Linux kernel running on the XScale. The first, which we term the **softnic** approach, is to have the MEs emulate a high end server NIC [11], while leaving the networking stack

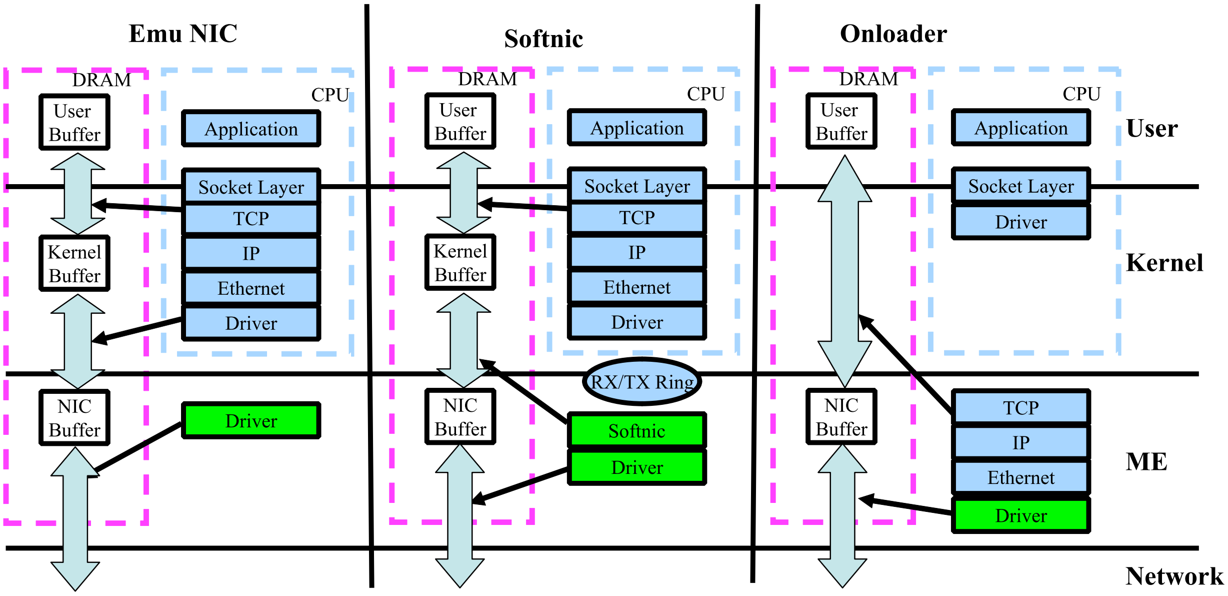


Figure 2.2: Architecture Comparison

on the XScale unmodified. Our second system is an onload engine that moves the networking stack to the MEs, and only performs high level interface functions on the XScale. Both systems either enhance or replace the kernel's networking stack and support the sockets interface for communicating with user programs.

2.4.1 Software NIC

Figure 2.2 contains an illustration of the **softnic's** architecture. To transmit a packet from the **softnic**, a user program calls the *sendmesg* system call. The upper, unmodified layers of the network stack in the Linux kernel will copy the data to be sent into a kernel space buffer, determine the interface the packet should be sent out on, and add headers. At this point, the fully formed packet is passed to the driver layer code, which is where our softnic modifications take over. The driver code places the buffer on a ring to the MEs for transmission. The kernel on the XScale is now finished with this packet and can go on to process the next one. An ME is constantly polling this ring (a hardware controlled scratchpad ring) for work. When it dequeues a packet buffer, it copies the contents into an internal buffer and raises an interrupt, letting the Xscale know it can now free that buffer. The internal buffer is then passed to another ME in a pipelined fashion for transmission. When a packet arrives at the MEs in the **softnic**, its checksum is verified, and it is copied into a kernel packet buffer, a pool of which has been preallocated for the MEs' use. The filled buffer is put on a ring for delivery to the XScale, and an interrupt is raised. The interrupt handler on the XScale will turn off interrupts, pull packets off the receive ring, and enqueue them for processing by higher levels in the kernel. Interrupts are re-enabled when the receive ring has been emptied. This is the adaptive polling technique. Control devolves to the unmodified Linux stack and a soft interrupt is raised, invoking the protocol processing code.

2.4.2 Onloader

Figure 2.2 also illustrates the organization of the **onloader**. When *sendmsg* is called in the onloader, the kernel prepares the user buffer for DMA and signals the MEs that a buffer is ready for processing. The MEs copy data directly from the user buffer into an internal buffer. The MEs then add headers and transmit the packet.

When a packet arrives at the **onloader**, an ME verifies the checksum, examines the headers, looks up the control block for that connection, and enqueues the packet for the proper connection. An interrupt is raised only if there is an idle process waiting for that packet to arrive. The only work the Xscale needs to do is to notify the waiting process that a packet is now available.

Our onload engine currently only supports UDP over IP. We believe that our results will also apply to TCP, as most of the OS infrastructure, such as interrupts, DMA, sockets interface etc. are common between them. The only major difference is the protocol processing step, which is a demonstrably small component of packet processing [14].

2.4.3 Emulated NIC

To determine how well the **softnic** and onload engine accelerate networking, we compare them to a base case wherein the MEs perform the minimum possible work to get packets to and from the MSF and most tasks are left to the XScale (called the **emu nic** in the graphics). The left side of Figure 2.2 illustrates the organization of the emulated nic. The **emu nic** corresponds to a low end NIC in a desktop system. The main difference between the **softnic** and this base case is that the driver code on the XScale must compute checksums and do all data copies between kernel buffers and device buffers. Additionally, interrupt handling is more expensive, because interrupts are raised for every packet on reception instead of adaptively polling after the first one.

2.5 Results

This section describes the experiments we ran to evaluate the effectiveness of the onloader in accelerating networking operations.

2.5.1 Experiment Setup

Our hardware setup consists of an IXP2350 system, connected through a gigabit Ethernet switch to a PC running Linux 2.4.19. The PC sends packets to the IXP to test the IXP's receive throughput, and receives packets from the IXP to determine the IXP's send throughput. We have determined by sending packets between two PCs that the PC does not represent a bottleneck.

As will be seen, applications executing on the 900 MHz XScale processor cannot receive or transmit packets at rates greater than 500 Mbps. While the packet processing code on the MEs can sustain approximately 2 Gbps, this rate cannot be delivered to the XScale and the applications it hosts. The main challenge facing the end-host system is data copying. Data not only has to be copied from the network into internal buffers and out again, but also into and out of user buffers within the system. As we will see, the cost of this is due not only to moving bytes, but also to pinning and aligning with virtual pages. Normally, router applications implement their fast path on the MEs alone and use the XScale for exceptions, but since we must interface with user programs on the XScale, we incur the overheads of sharing it with other OS functions, such as timer interrupts, or task scheduling.

While the XScale on the IXP2350 cannot perform end-host network processing tasks at gigabit rates, we note that our goal is not absolute performance, but to validate our idea that small, simple, efficient cores attached to a general purpose processor can accelerate network processing. Hence, our use of the emu nic as a base case for performance.

We ran our experiments using the Iperf benchmark [37] in UDP mode. In server mode, Iperf waits for a client to connect, and counts the number of bytes received until a termination packet is received. A timestamp is taken after reception of the first packet and reception of the termination packet for determining the achieved throughput. The client program sends fixed sized packets for a given amount of time, followed by a termination packet, and keeps track of the number of bytes sent and the elapsed time. This is a test of throughput in a bulk data movement application.

2.5.2 Receive Path

Figure 2.3 shows the achievable receive throughput for the 3 cases. We can see that both the **softnic** and **onloader** are clearly superior to the base case. For large packets, a nearly 10-fold improvement is seen. This is mainly due to the MEs' superior ability to move memory from buffer to buffer. The base case NIC suffers from the XScale's more limited bandwidth when copying between two buffers. Furthermore, while the number of data copies is the same between the base NIC and the **softnic** (from the MSF to an internal buffer, to a kernel buffer, to a user buffer), the **softnic** handles the copy from the internal to kernel buffer asynchronously on the MEs. The

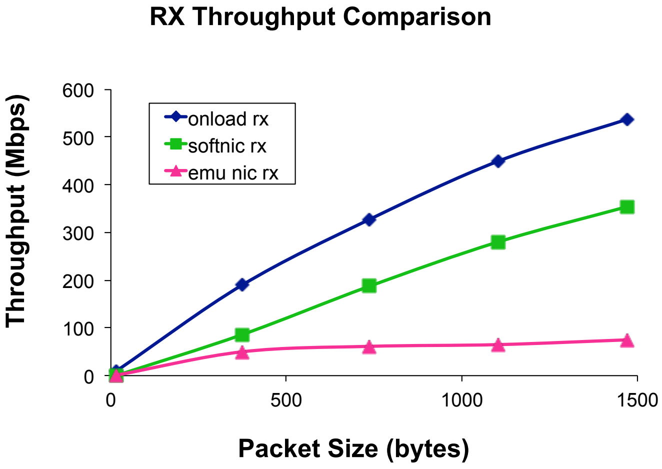


Figure 2.3: Receive Throughput Comparison

onloader avoids the copy to a kernel buffer altogether and copies data directly from its internal buffers into the user program's buffers.

Figure 2.4 shows the percentage improvement of the **onloader** over the **softnic** for different packet sizes. Between the **softnic** and the **onloader**, the **onloader** has superior receive performance, with throughput increase between 100% and 40%. This difference is especially true for smaller packets, where per packet overheads, such as header processing and control buffer lookups, dominate execution time. The main reason for the improvement is that the **onloader** can asynchronously receive and enqueue packets while the XScale can be dedicated to other tasks, such as running the userspace benchmarking program. For the **softnic**, the XScale must split its time between packet processing and other tasks. With larger packets, the per byte costs of checksumming and data copying are dominant, and as this is done on the MEs in both the **softnic** and **onloader**, the difference between them becomes quite small, about 20%.

Figure 2.5 shows the throughput of various onloader receive components. The top line shows the sending rate of the PC. The first set of bars is the receive throughput of the driver and receive blocks of the onloader, with no user program consuming the received packets. These blocks receive packets from the network, move them into an internal buffer, verify checksums, parse the headers, look up control blocks and

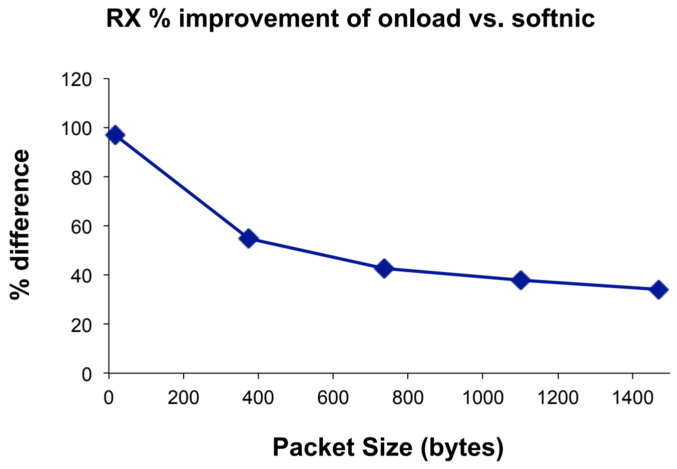


Figure 2.4: Receive Percentage Difference

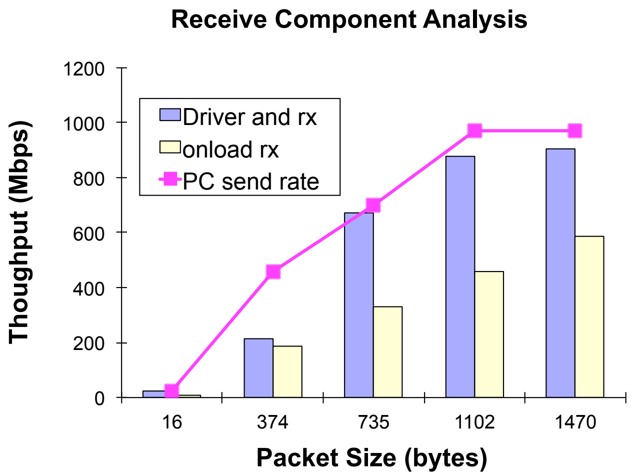


Figure 2.5: Receive Components

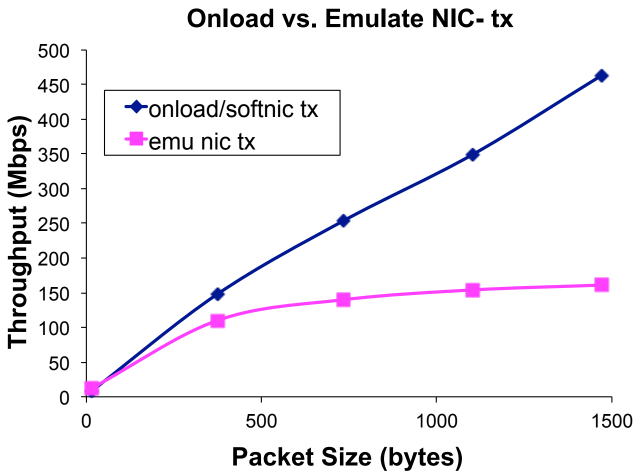


Figure 2.6: Transmit Throughput

enqueue the packets for future reference. They achieve a receive throughput very close to the sender's sending rate. The second set of bars shows the throughput of the entire **onloader** system, including the driver and receive block, as well as an XScale component that calls the *recvmsg* system call and moves the packet payloads into a user buffer. As the driver and receive blocks receive packets about as fast as they are being sent, we must conclude that the movement of data into user space using sockets is our receive bottleneck.

2.5.3 Transmit Path

On the transmit side, the **onloader** and **softnic** are again superior to the base case, as demonstrated in Figure 2.6. The main reason is because the **onloader** and **softnic** take advantage of the MEs' superior ability to move memory, whereas the base case is hampered by the XScale's limited memory throughput. There is no clear advantage for either the **softnic** or **onloader** on the transmit side, as the main bottleneck here is memory copying and checksumming, which are offloaded to the MEs in both cases. As with the receive path, the **softnic** on transmit has the advantage of asynchronously performing a DMA from kernel buffers to ME buffers on the MEs while the onloader copies directly from user buffers to internal ME buffers.

2.6 Design Improvements

Our experience designing the **onloader** has demonstrated some aspects of the IXP design that should be modified in a system designed for onloading. These observations apply to any on-chip I/O acceleration technique. One such weakness is the lack of a Memory Management Unit (MMU). In order for the MEs to copy data directly to and from user space buffers into internal buffers, achieving the equivalent of zero copy semantics, the kernel on the XScale must first ensure that all pages of the user buffer exist in memory, walk the system's page tables to find their locations, and clean or invalidate the cache in order to keep consistency. This is very inefficient; in early implementations, we found this cost to account for a third of the per packet overhead. In order to get around this problem, we cached previous mappings of user space buffers for each connection, so that if a program reuses the same buffer (as the Iperf benchmark does), this costly overhead does not have to be incurred again. This resulted in a 1.5X increase in throughput for the receive path, which is reflected in our numbers for the previous section. Without this optimization, the onloader is actually slower than the softnic, despite eliminating one buffer to buffer copy. Giving the MEs a Translation Lookaside Buffer (TLB) would effectively do the same job in hardware, without having to incur page walking overheads on the XScale.

Another optimization would be to have the MEs participate fully in the XScale's cache coherence protocol. The ability of the MEs to push data into the L2 on writes to DRAM was a big improvement for both the **onloader** and **softnic**. If, instead of using the push feature (which is optional), the XScale were to invalidate those addresses in its cache and reload the data from memory, our experiments show that the softnic would suffer a 35% degradation in performance. As of the current model IXP2350s, this is the only feature available for cache coherence. A full coherence protocol, which would include letting the MEs snoop the XScale cache on a memory read, would prove advantageous for the transmit path.

One problem with supporting the sockets interface is that the system has no control over where the application allocates its user buffers. Thus when moving data into these buffers, the DMA mechanism must contend with both crossing page boundaries and DRAM word alignment. For DRAM alignment, a read-modify-write may be necessary to avoid overwriting other data that shares a given DRAM word with the user buffer.

2.7 Conclusion

The architecture proposed in this section is a set of hardware and software changes that we believe represent the best way to preserve traditional socket programming semantics in modern multi-core systems. In the next section, we examine strategies for parallelizing applications on existing hardware and apply those lessons to the design of a network API targeted at writing efficient, parallel networking applications on multi-core processors.

Chapter 3

Parallelization of Snort

The Snort [42] intrusion detection program is a popular tool for securing networks using deep packet inspection to detect the signature of malicious packets. For Snort to be effective, it must be able to keep up with increasing line rates. Future processors will improve performance through the addition of more cores, and high performance programs such as Snort need to be parallelized to take advantage of these newly available resources.

We parallelize Snort by running a full copy of the Snort detection engine on each core in the system. Packets of the same flow are processed in order on the same core and parallelism is achieved by processing packets from different flows in parallel on different cores.

We further contribute a study of the efficacy of a static flow pinning scheme under realistic scenarios. We test our parallel Snort implementation against real world packet traces collected from multiple sources, including the internal and external Intel web servers, and publicly available traces of connections between university networks and the internet core from the National Laboratory for Applied Networking Research (NLANR). Results from the Intel servers are not presented directly for privacy reasons, but were used to confirm results obtained using the other traces. The NLANR traces are no longer available on the web, but this paper [29] gives a good overview of their characteristics. The lessons learned from this case study will be applied in the following chapters to the design of an API for writing efficient parallel networking code.

3.1 Parallel Snort

We parallelize Snort by replicating its functions across multiple cores and processing multiple packets from different flows in parallel on different cores. Packets in the same flow are processed sequentially on the same core to preserve in order delivery and limit the sharing of per flow data across cores, which reduces lock and cache contention between cores. We use a two stage pipeline to process packets- the first stage, running in a dedicated core, receives packets (using libpcap), performs classification on those packet and passes them to the second stage. The second stage contains the rest of Snort's functionality, such as stream reassembly, regular expression matching, and event logging. This second stage can be replicated in multiple threads, with each instance pinned to a particular CPU core. Flows are pinned to a particular thread in order to increase cache locality and reduce context switching overheads. Flow pinning also allows us to eliminate locks for flow specific data structures. Another optimization is the per thread packet pool- using separate buffer pools increases cache locality and allows the queues to be implemented as lockless ring buffers. Furthermore, using preallocated buffers improves performance by eliminating expensive *malloc* calls from the fast path. While there have been other attempts to parallelize Snort [6] [27] [46] [7], we are aware of only one other that does so in a flow aware manner [45]. Their methodology differs from ours in that they do dynamic reassignment of flows and and do not have a thread dedicated exclusively to flow classification. It is hard to compare our results to theirs, as we use different packet traces, different versions of Snort and different hardware. However, they observe a roughly 3x speedup when moving from 1 to 4 threads, which, as demonstrated in the next section, is what we observe as well.

3.1.1 Experimental Setup

Our experiments were run on an 8 core Xeon system (dual quad core CPUs), with packet traces read off a disk to avoid having the network become a bottleneck, since we are interested in the performance of the Snort program, not the network stack. Furthermore, dropping packets can cause major changes in Snort's behavior and reading the trace from a file allows Snort to throttle its own input rate. We measured

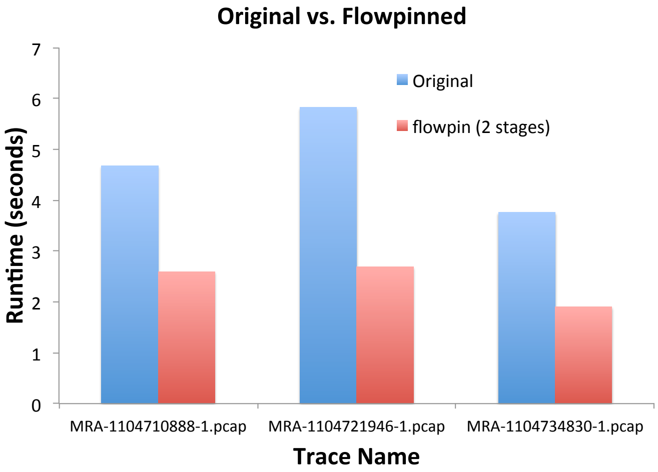


Figure 3.1: Flow Pinning vs. Simple Threading

the total time Snort requires to process a trace and used that as the metric for comparison.

Our traces were collected from multiple sources, including the internal and external Intel web servers, and connections between university networks and the internet core (NLANR), described earlier in this chapter. We believe these to be a good sampling of real world workloads likely to be encountered by Snort. Since these traces only include packet headers, synthetic packet bodies were inserted for testing. These packet bodies contain random data and are not meant to simulate any particular attack signature.

3.1.2 Evaluation

Our first experiment was to examine the efficacy of flow pinning. For this we had 2 versions of Snort- one that does flow pinning, and one that does not and has locks inserted for the access of per flow data structures. The result, as demonstrated in Figure 1, is that flow pinning is an important and effective optimization, though the level of effectiveness depends on the particular workload. This being the case, we focused our work on the flow pinned version of Snort. Next, we tested the scalability of this scheme by running Snort over several workloads using different numbers of cores. Figure 3.2 illustrates the scaling of 2 stage Snort for one of the workloads

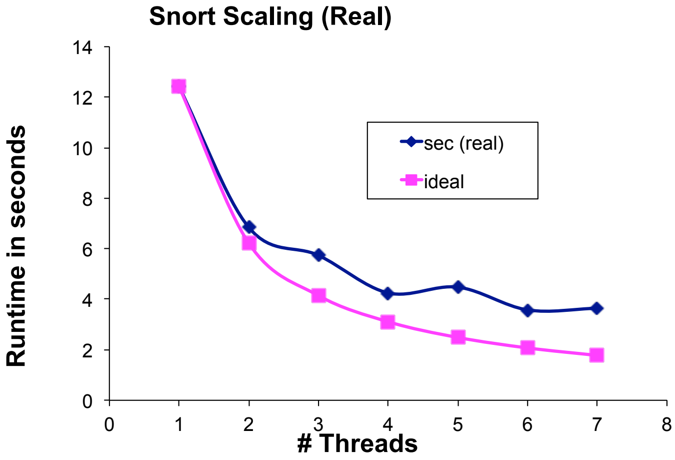


Figure 3.2: Two Stage Snort Scaling vs. Ideal

(MRA-1104710888-1), along with a curve representing ideal scaling (i.e. time to run Snort using one core divided by N). As more cores are added, Snort's performance deviates further and further from ideal. For some traces (not shown), performance even degrades above a certain number of cores. Table 3.1 shows the data used to make that graph as well as well as the actual % difference from ideal.

Table 3.1: Two Stage Snort Scaling Data

Num Threads	**Seconds**	**Ideal**	**% difference**
1	12.4	12.4	0
2	6.8	6.2	-9.6
3	5.7	4.4	-29.5
4	4.2	3.1	-35.4
5	4.5	2.5	-80.0
6	3.6	2.1	-71.4
7	3.6	1.8	-100.0

3.2 Scaling

We found that failure to scale as expected was caused by an uneven distribution of work among the threads. Some packets take much longer to process than average,

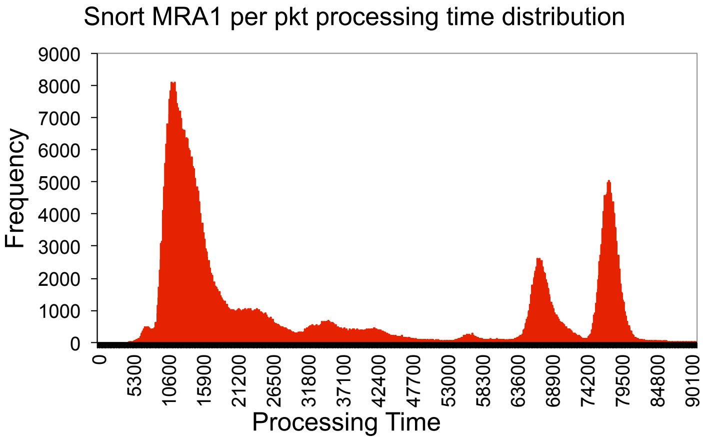


Figure 3.3: Packet Processing Time Distribution

and those packets can cause a single thread to be continually busy while the others clear their queues and sit idle. This is true even if the number of packets processed by each thread is similar. Figure 3 shows the distribution of time required to process individual packets in Snort when only a single thread is used. Using only one thread eliminates variables such as lock contention and competition between threads for resources (buses, L2 cache space, etc). As can be seen from the distribution, most packets take around 12000 clock cycles to process, but there are plenty of packets that take more, with clusters around 68000 and 77000 clock cycles. Not shown on the graph are 3 additional packets that take several orders of magnitude more processing time. These longer packets might cause a single thread to become backed up and block the entire program, especially if they occur in bursts and are correlated on specific flows. We believe the latter behavior to be likely, but have not tested it.

In order to confirm our suspicion that uneven packet times are the cause of our poor scaling, we wrote a simulator that takes a trace of packet runtimes from the two stage Snort using 1 process thread, and simulates the queuing behavior. This allows us to eliminate lock contention and other possible factors. We wanted to see if we could replicate the scaling issues in the simulator, and if clamping the runtimes of all packets to not exceed a maximum value, we could improve the scaling.

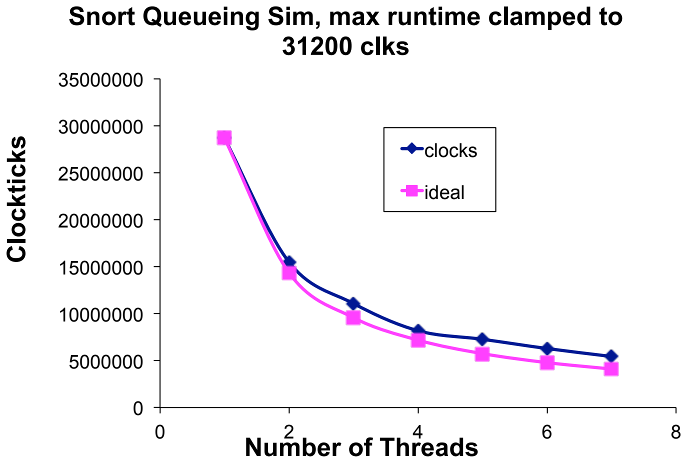


Figure 3.4: Queuing Simulation Scaling with Runtimes Clamped to 31200 Clocks

Table 3.2: Queuing Simulation Scaling with Runtimes Clamped to 31200 Clocks

Num Threads	Clocks	Ideal	% difference
1	28693218	28693218	0
2	15475780	14346609	-7.871
3	11071555	9564406	-15.758
4	8175468	7173304	-13.971
5	7264940	5738643	-26.597
6	6282468	4782203	-31.372
7	5434540	4099031	-32.581

We ran our simulator using an unmodified packet trace and received results similar to what we saw in practice. Next, we clamped the max runtime for any packet to be 31200 simulated clock ticks (which are NOT directly convertible to real clock ticks as they have been scaled to speed up simulation runs). Looking at the packet processing time distribution, 31200 is at the high end of the first peak in the graph, and we consider anything above that to be an excessively long processing time. Thus, clamping the runtimes to 31200 clocks should effectively eliminate the effects of excessively uneven packet processing times from the simulation, while maintaining the same number of packets and their distribution to the threads. The result, shown in Figure 3.4 is scaling that is much closer to ideal. For exact numbers, compare the % difference from ideal in table 3.2 to the one in table 3.1. We feel that the results of this simulation study confirm our theory that uneven packet processing times are the main reason for Snort's lack of scaling. If poor packet distribution is indeed a problem, it would seem to be a lesser one.

3.3 Lessons

The experience of programming and benchmarking Snort has taught us several lessons that we will apply to the design of a parallel network programming API in the following chapters.

First is the efficacy of flow-pinning in reducing lock contention in complex, stateful programs. Our experiments clearly show that this is an effective strategy and should be incorporated in future applications.

Second, the fact that Snort's performance is being gated by the relatively few exceptional packets is an interesting result. Head of line blocking by exceptional packets becomes a major concern with the static scheduling scheme currently in use. In the next chapter, we develop new scheduling strategies that might behave better under these circumstances. Unfortunately, we were unable to apply them to the parallelized Snort implementations we have been using throughout this chapter, as they are owned by Intel and were no longer available to us at the time of the following experiments.

Chapter 4

Software Router API

Network devices are becoming ever more complex. The demand for new capabilities in routers has lead to an interest in programmable network devices that are capable of high speed packet processing. However, the traditional programming tools for general purpose processors are not designed for the specific and demanding task of network packet processing. In our view, no current programming environment provides the ideal set of characteristics for a networking device. These include good performance, reuse of familiar languages, ease of use, and backwards compatibility. Because the needs of the network programming community are not being met by existing solutions, we see an opportunity to provide a programming environment that will be beneficial to network operators, system vendors, semiconductor vendors, and software developers alike.

In this chapter, we will present our API for writing high performance packet processing applications that run on general purpose processors.

4.1 Requirements of a Framework

The goal of this chapter is to present a framework for writing network centric programs. We believe the most important considerations for such a framework are performance and backwards compatibility, which we discuss in this section.

Table 4.1: Processor Characteristics

Processor (NP=Network Processor GP=General Purpose Processor)	Clock (MHz)	CPU Power (W)	Chipset Power (W)	Packet I/O (Gbps)	Mem I/O (Gbps)	Processor Cores	Core Issue Width	Peak BIPS	Peak BIPS/W
Cisco SPP (NP)	250	35	0	192	175	188	1	47	1.34
Intel IXP2855 (NP)	1500	27	0	25	121.16	16	1	24	0.89
Cavium Octeon CN5860 (NP)	1000	40	0	25	102.4	16	2	32	0.8
Raza XLR 732	1000	32	0	25	230.4	8	1	7	0.25
Cavium Octeon CN3860 (NP)	600	30	0	25	102.4	16	2	19.2	0.64
Intel Quad-Core Xeon 5300, Intel 5000P chipset (GPP)	2330	80	30	0	85.6	4	4	37.28	0.34
AMD Dual-Core Opteron 1218 HE (GPP)	2600	65	0	192	85.6	2	3	15.6	0.24
Niagara 2 (NP)	1400	84	0	40	307.2	8	2	22.4	0.27

4.1.1 Performance

Modern processors have enough raw performance to process packets at high rates. Even general purpose processors of the newest generation, such as a 4-core Nehalem, can process minimum sized packets at close to line rate for gigabit connections. Table 4.1 shows the peak performance (expressed in billions of instructions per second) for a variety of NPs and GPPs (data from [22] [35] [20] [2] [24] [15] [4]) . NPs tend to use less power and are designed to be able to achieve that peak, but GPPs tend to have the best peak performance, leading us to believe that they should be able to compete with NPs if properly harnessed. A poor software layer can significantly reduce performance. A major goal of a good framework should be that it can be easily implemented on a number of systems without unduly strangling performance. For example, a major shortcoming of the NP-Click [47] project is that small-packet performance is greatly reduced compared with that on an application written in the IXPs native C-derivative.

Our proposed API, the Network Runtime Environment (NRTE) uses a pipelining approach that has several advantages for getting the most performance out of a general purpose processor. Pipelining allows us to reduce the cache footprint on each core by reducing the working set of each core. Secondly, pipelining allows us to process multiple packets in different stages in parallel. Finally, the use of flow-pinning takes advantage of cache locality by keeping per flow data structures on the same core and its associated cache. This reduces cache thrashing and lock contention.

4.1.2 Backwards Compatibility

We expect future generations of networking devices to get a performance boost from additional threading and fixed-function accelerators, and the ability to take advantage of these new features automatically is paramount. How this can be done is an open question. Some attempts have been made to find ways to automatically map program components to hardware threads [16] [12]. None of these solutions is perfect. First, there is a tradeoff between static mapping at compile time and automatic remapping at runtime. The former obviates the need to have a runtime system that can remap program components, which can potentially require significant overhead. The latter would be better at adapting to changing workloads. As demonstrated in [39] the ability to adapt to different workloads can significantly improve performance in the face of changing workloads.

Future proofing is difficult; if we make the assumption that future performance comes from greater parallelism, the proper way to exploit that parallelism automatically is not clear. One possibility is to organize programs in a pool of worker threads, and to scale up the number of program threads as the available number of hardware threads increases, but not all applications are amenable to this type of parallelization.

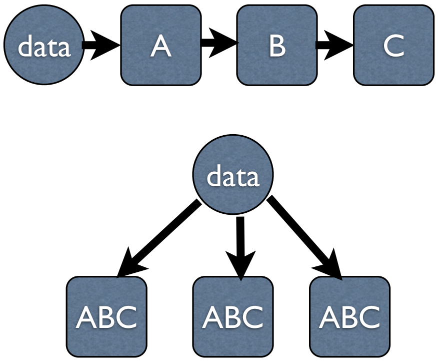


Figure 4.1: Pipeline (top) and Pool of Threads (bottom)

Programs that use pipelined parallelism and distribute their stages among multiple hardware threads can only scale to a point; once the number of hardware threads exceeds the number of explicit stages in the program, these applications will no longer automatically benefit without programmer intervention. Figure 4.1 illustrates the two different types of organization. At the very least, programmers must either change their code to create more duplicate threads, or write extra code to allow their programs to adapt automatically, but this is cumbersome and done on a case-by-case basis. The NRTEs solution is to duplicate pipeline stages at runtime, which allows it to use both strategies in a straightforward manner without the need to rewrite or recompile the application.

4.2 Existing Frameworks

There exist many APIs and programming frameworks that solve some aspect of the problem we are examining. We examine their characteristics in this section.

RouteBricks [19] defines a vision for using a cluster of PCs to process up to 35 Gbps of traffic in a software router. We aim at more modest targets using a single PC. Routebricks exploits parallelism in multi-core architectures by having each core dedicated to performing network processing for a single NIC queue rather than pipelining. They are targeting applications where there is little processing at each node and network I/O is the bottleneck. We are more concerned with applications that are dominated by application processing.

The Click [30] framework provides a library of predefined elements for common networking tasks that can be composed into a complete application using the Click language. Additional functionality can be added by writing new elements in C++ that conform to the Click framework. One disadvantage is that it requires a programmer to learn the Click language to write applications. Another disadvantage of Click is that the framework for writing new elements is written in C++, which is not usually available for embedded platforms such as network processors. While Click configurations are potentially portable if a Click implementation that includes the elements used in the configuration is already available for the target platform, the implementation of new elements is not. Implementing new protocols (for example, [49]) not supported by existing Click elements is difficult to do in a portable fashion. The NP-Click [47] project created a Click implementation for the IXP network processors, but the underlying element implementations were rewritten in IXP-C, and the interface for writing new elements is incompatible with the original Click.

There are various solutions for easing the programming of NPs that abstract away some of the difficulties of programming in a low-level environment. These include Shangri-la [12] and the Intel auto-partitioning compiler [16]. These are C-based solutions, but they both run only on Intel's IXP network processor. Shangri-la's Baker programming language is platform-independent, but it is not truly portable because no implementations exist for other platforms.

Another proposed solution is NetVM [17], a virtual machine for network processing. NetVM attempts to define a virtual machine with its own virtual instruction set. An interpreter or compiler can take this generated byte code and run it on a specific platform. The authors define an architecture for this virtual machine, but not a programming model. Moreover, the prototype performs rather poorly. It took 2236 clock cycles to perform an IPv4 filtering task that took a Berkeley Packet Filter implementation 124 clocks to perform.

Recently, Juniper began offering third parties the ability to program their routers through their Partner Solution Development Platform (PSDP) [36], which provides a framework on top of the JUNOS operating system running on their routers. This solution apparently provides a uniform framework for all Juniper routers, but there is not much information publicly available. However, it demonstrates that companies are beginning to see the advantages of opening up their router platforms to third-parties.

Finally, projects such as XORP [28] pursue a complementary goal. XORP is an interface for control-plane processing, which runs above the data plane. XORP can work using different data-plane implementations, including Click. It could also potentially implement the data plane using the interface we will introduce in this chapter.

4.3 NRTE

The Network Runtime Environment (NRTE) is our implementation of a multi-core oriented, network programming environment. It is implemented as a C library, which presents programmers with a familiar programming language that is portable across many platforms. It is explicitly designed to allow programmers to expose as much parallelism as possible in their programs while leaving the details of mapping to the underlying hardware threads up to the runtime.

The NRTE requires the user to handle parallelism. It is the programmer's responsibility to break the application into stages and to make these stages thread-safe. The application is organized as a pipeline with stages communicating with each other over queues. The runtime is responsible for mapping these stages onto the underlying hardware. Additional parallel computing resources are taken advantage of by duplicating stages and splitting the incoming packet flows among these duplicates. The user specifies the hardware mapping using a control program.

The NRTE provides two types of stages: explicit and implicit. The explicit stages are threads under explicit user control; they launch and run to completion much like a thread created using the pthreads library. Implicit stages are registered as callback functions on an associated queue. The registered function is called by the runtime to process elements on that queue. The instantiation of implicit stages is left to the runtime; if the user indicates that the implicit stage is safe to duplicate (i.e., is thread-safe), the runtime can create duplicates of that stage to run on multiple hardware threads. Packets are distributed among the duplicates using flow-pinning. The user specifies the flow definition via a classification function. The runtime uses this function to send packets of the same flow to the same stage, so they can be processed in order and take advantage of cache locality for flow-specific data structures. Parallelism is achieved by processing different flows on different threads. The use of implicit stages in this manner allows us to scale the application to take advantage of increasing numbers of cores on future processors.

The NRTE's strategy for dealing with multi-threading differs from Click's. Click creates a task list for its configuration and load-balances the tasks across cores. This does not take into account caching effects, and it has a fixed amount of parallelism. If there are more hardware threads than tasks, Click cannot make use of them. By repeatedly duplicating the bottleneck stage(s), the NRTE actually creates parallelism, which can potentially scale to as many threads as are provided by the hardware.

We confirmed the usefulness of flow-pinning by creating an application that mimics the access pattern of stateful packet-processing applications. This application receives incoming elements and accesses and modifies state associated with that elements flow. We measured the time it took the application to process a fixed number of elements; we found that when flow-pinning was used to distribute the incoming data, the application ran in 12.04 seconds. When the incoming data was distributed without regard to flow, it took 16.19 seconds. This is due mostly to cache-thrashing. Our micro-benchmark does not include the cost of locking, which would likely be needed to prevent data corruption in a real application. Including this would further improve the flow-pinning results because it eliminates the lock synchronization overheads incurred without flow-pinning when multiple threads attempt to access the same flow state.

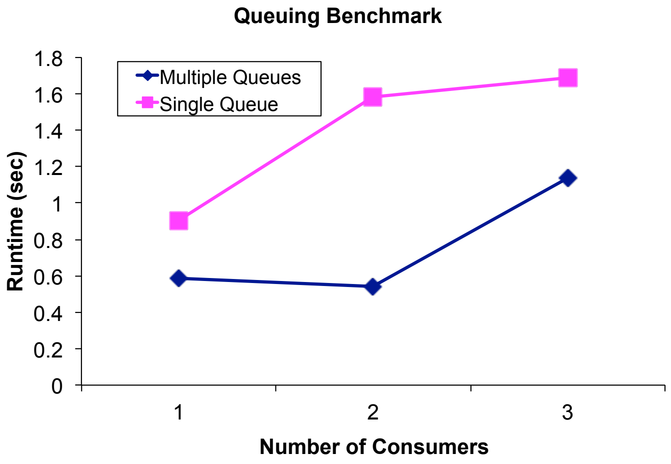


Figure 4.2: Queuing Benchmark

When a stage is duplicated, the underlying queues used to connect that stage to the rest of the pipeline are also duplicated. This is done to prevent synchronization overheads that would result if there were only one underlying queue with multiple replicas trying to read from it. When an implicit stage is duplicated by the runtime, there will be more than one consumer on that stage's queue. A simple way to maintain integrity of the underlying queue would be to use mutual exclusion locks to allow atomic access to a single producer or consumer at a time, but we observed that this quickly became the program's bottleneck. Instead, by implementing the logical queue abstraction as a set of point-to-point queues with a single producer and a single consumer each, we remove this necessity. The underlying physical queues do not require locks. Figure 4.2 shows a micro-benchmark we constructed comparing the use of the NRTE's queuing strategy of single consumer/producer circular buffers with an implementation where a single queue (still a circular buffer) is shared among multiple consumers, requiring synchronization with locks. The program has a single producer enqueuing items onto a queue and we measured the time it took for the consumers to dequeue all of them. The producer was producing while the consumer was simultaneously consuming. We varied the number of consumers from one to three, mapping a single producer or consumer to each core in our test system until we ran out of cores. The data points shown are representative of the observed performance but are not deterministic. The data show that the single-queue implementation, which requires lock synchronization for each enqueue or dequeue, performs far worse than the multiple-queues implementation. In addition, as we vary the number of consumers, the single-queue implementation scales poorly. The multiple-queues implementation scales well; no change in performance is observed between the one- and two-consumer cases. When moving to three consumers, performance degrades, but not for queuing reasons; the third consumer is placed on a core that shares an L2 cache with the producer, resulting in cache-capacity problems.

The NRTE also includes a packet abstraction and libraries to deal with common packet-processing tasks. Packet-handling abstractions allow the user to write platform-independent code, so that running NRTE-derived programs that must run in different environments does not require extensive rewrites. For example, in different situations, the same application may be run as a user space program that manipulates packets using sockets; or as a Linux kernel module that directly manipulates packets in *skbuffs*, the kernel's data structure for storing packet and meta data information. Libraries that provide common functions such as check-summing or efficient algorithms for longest prefix match will allow programmers to concentrate on what is unique to their programs rather than reinventing the wheel. Finally, putting efficient implementations for common data structures into the framework provides both the convenience and the ability to take advantage of underlying hardware support without programmer intervention. An example of this is data encryption. Hardware support exists for this on specialized platforms such as the Tolopai [3]. On older x86 platforms this is not present, but efficient software implementations, will be substituted.

Table 4.2: NRTE API Summary

Function Name	**Description**
rte_register_explicit_input_function	Create a stand-alone thread
rte_register_queue	Create a queue and register it's handler function
rte_enqueue	Enqueue an element to a queue
rte_start	Start runtime after all elements are registered

Table 4.2 contains a summary of the NRTE API. A complete listing is given in appendix A. We will illustrate the workings of the NRTE with an example. Our example begins with the following code snippet:

```
main() {
    a = A();
    b = B(a);
    C(b);
}
```

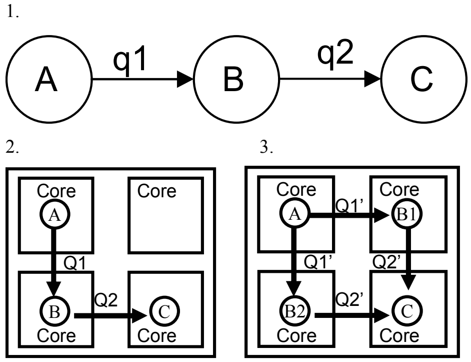
Where A(), B(), and C() are functions that perform packet-processing operations. This can be turned into a pipeline, with dataflow illustrated in Figure 4.3 using the NRTE with the following code:

```
nrte_queue_id_t q1, q2;
void A(void *ignored)
{   //Do A's work, then send on
    uint64_t A_elem;
    nrte_enqueue(q1, A_elem);
}

void B(uint64_t stage_id, unsigned int flow_id, uint64_t A_elem)
{  //Do B's work, then send on
   uint64_t B_elem;
   nrte_enqueue(q2, B_elem);
}

void C(uint64_t stage_id, unsigned int flow_id, uint64_t B_elem)
{  //Do C's work
}

unsigned int flow_classifier(uint64_t elem)
{   //dummy function. Simply returns input
    return (unsigned int)elem;
}
main()
{
    //Create the flow graph in the NRTE
    q2=nrte_register_queue(C, flow_classifier);
    q1=nrte_register_queue(B, flow_classifier)
    nrte_register_explicit_input_function(A);
```



1. **Programmer-specified application structure.**
2. **Pipeline can be mapped to 3 cores.**
3. **Same pipeline can be mapped to 4 cores without modifying the program. Queues into and out of thread-safe replicated stage are automatically duplicated so each has a single producer and consumer, thus avoiding locks.**

Figure 4.3: NRTE Dataflow

```
    nrte_start();
}
```

This code creates an explicit stage to run the first calculation, A(), and two implicit queues to calculate B() and C(). When A() finishes its calculation, the result is enqueued to the second stage, which performs calculation B(), which in turn passes its output to stage C(). A dummy flow classifier function is defined, which will be called if the implicit stages are duplicated to help the runtime decide which copy to pass it to. The main function is responsible for registering the stages with the runtime and then kicking things off by calling rte_start(). Once this happens, control passes to the runtime, which will start the explicit stage, A(), and run the implicit stages B() and C() when items appear on their respective queues. The three stages would most likely be run on three different hardware cores or threads.

Figure 4.3 shows the mapping of the logical pipeline to hardware. The first configuration maps a single copy of each stage to a separate core. The next panel illustrates what happens if the runtime decides to replicate the second stage, B. The stage is copied and the second copy mapped to another hardware thread. The queues associated with it are also duplicated; stage A will see one logical queue, (q1) but the underlying implementation is two single-producer, single-consumer queues (the q1s). The runtime takes care of deciding on which queue to put an element enqueued from A. The same is true at the other end; each copy of B has its own queue to C, although to the single thread running C, it looks like a single logical endpoint. The use of separate point-to-point queues in the underlying implementation allows these queues to be implemented efficiently and without locking, which, as we demonstrated earlier, can quickly become a bottleneck if there are multiple producers and/or consumers.

Note that the duplication and mapping of the stages is done at runtime by a control program and is not expressed in the application itself. This makes it easy for the programmer to make their code independent of the number of cores on the system; mapping to hardware is done at runtime. Furthermore, the programmer can experiment with different mappings to find the optimal mapping for a given platform or workload.

This section outlined our vision for a high performance, multi-threaded networking API. In the next section, we present an evaluation of our prototype implementation.

4.4 Performance Evaluation

We have implemented a prototype of the NRTE that runs on x86 Linux systems. This initial prototype implements the NRTE threading model and packet handling abstractions, but requires the user to manually map stages to cores at runtime using a control program; automatic scheduling will be discussed in a later section. This section demonstrates that the NRTE prototype demonstrates the qualities of a good API that we outlined above without sacrificing performance.

4.4.1 Test Setup

Our test system uses dual, dual core Xeon processors (a total of 4 cores), and a 4 port e1000 network card, running Linux with a 2.6.20.1 kernel. Another machine using the Linux kernels pktgen module is used as a traffic generator. Forwarding rates were measured at the receiving end, using a receiver that is capable of receiving packets faster than the test system is able to forward them. The pktgen module sends packets at a constant, adjustable rate. The source and destination addresses of the UDP packts being sent are incremented with each packet sent in order to send packets that will map to different flows. This setup is used for both the IPv4 Forwarder and NAT experiments presented below.

For our tests, we measured implementations of IPv4 forwarding and NAT, written using the NRTE against Click configurations for the same application running in user space. Click performs better running as a kernel module, but we envision our API will be most useful for complex applications written in user space where the networking overhead is not the bottleneck. The user space test is still meaningful as a comparison, and is the mode that must be used in platforms such as Planetlab [13], that do not allow programmers to run code in the kernel for security and isolation reasons. Environments like Planetlab are an important target, as they can be used to prototype new protocols, and if a portable API like the RTE is used in the prototype, moving to more high performance platforms will be easier

4.4.2 IPv4 Forwarder

Our implementation of the IPv4 forwarder is functionally equivalent to the Click router configuration we used. They use the same algorithm for longest prefix match (DIR 24-8 BASIC) [26], which performs efficient lookups in at most 2 memory accesses.

The NRTE forwarder uses two pipeline stages: an RX stage that receives and classifies packets, and a forwarder stage that does most of the forwarding work (checksum computation, route lookup). RX is an explicit stage; there is only one copy which is instantiated by the user. The forwarder stage is an implicit stage, which is duplicated three times, with packets split by flow (defined in this case simply by the destination address) among the copies. While no flow specific data structures are necessary in this application, flow pinning results in in-order packet processing for each flow, which has implications for protocols such as TCP. The duplication and mapping of the forwarder stage is done at runtime by a control program and is not expressed in the program itself, which allows future versions of the NRTE to automatically do this mapping

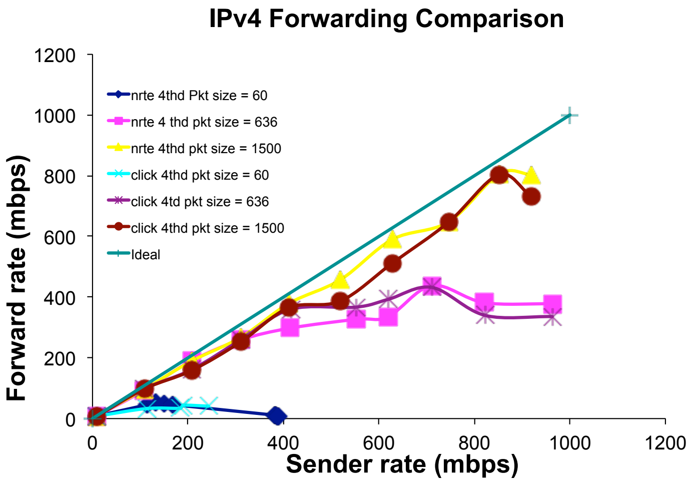


Figure 4.4: IPv4 Forwarding Comparison

without modifying the application code. In this manner, all 4 cores of our test system are utilized, with the decision of how many copies of the forwarder stage to create deferred until runtime.

We measured forwarding rates for each application for a variety of packet sizes and sender rates. The plotted data points are each from a single representative 60 second run and the details of the input traffic were described above. The results are shown in Figure 4.4. This data clearly shows that over a wide range of packet sizes and sender rates, the NRTE IPv4 forwarder performs as well or better than the Click configuration.

4.4.3 NAT

Our second test application is network address translation (NAT). We tested a click configuration using the IPRewriter class and an NRTE based implementation. The NRTE implementation used a single receive stage and three NAT stages that did most of the work. We tested the rewriting of IP destination addresses and UDP destination ports. Figure 4.5 shows the results, using a representative 60 second run for each sender rate tested. We can see that once again the NRTE and Click versions of the same application perform comparably.

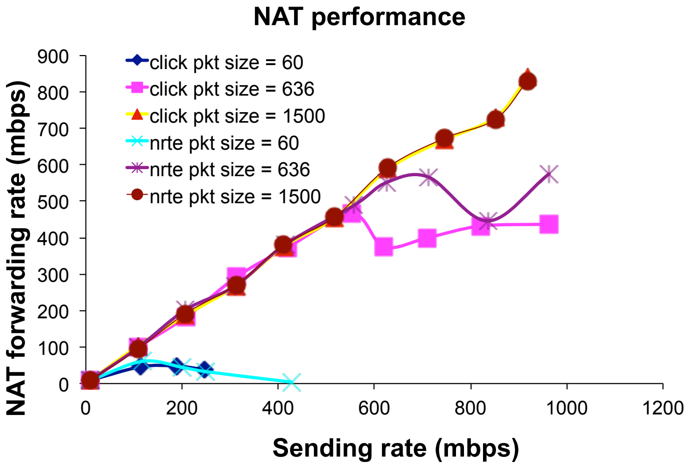


Figure 4.5: NAT Comparison

4.4.4 Snort

We have ported Snort [42], a popular open source intrusion detection program, to work with the NRTE. We divided Snort up into 4 functional blocks- a packet receive and classification stage run as an explicit stage, and three implicit stages- a preprocessor stage that runs Snort preprocessors such as TCP stream reassembly and portscan detection, a detect stage that performs regular expression matching and a logging stage that logs the results. The three implicit stages are written to be thread safe so they can be replicated. Packets flow from stage to stage in the order they have been listed, with packets in different stages running in parallel on different cores.

The pipelined nature of this application allows us to demonstrate one of the strengths of the NRTE. By leaving mapping decisions to runtime, we are able to test different pipeline configurations to find the best one for each workload. For some workloads, replicating the preprocessor stage improved performance; for other, replicating the detect stage was the better option. Having a control program that allows the programmer to decide the programs mapping to hardware at runtime, without rewriting the program, is thus an extremely useful tool.

The Snort experiments were run on an 8 core Xeon system (dual quad core CPUs), with packet traces read off a disk to avoid having the network become a bottleneck,

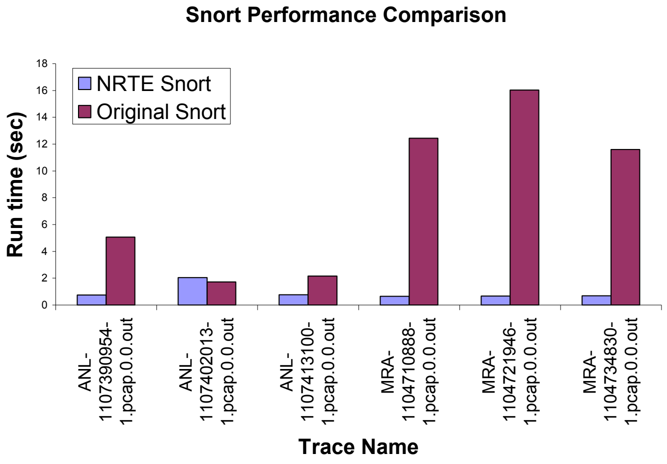


Figure 4.6: Snort Comparison

since we are interested in the performance of the Snort program, not the network stack. Furthermore, dropping packets can cause major changes in Snort's behavior and reading the trace from a file allows Snort to throttle its own input rate. We measured the total time Snort requires to process a trace and used that as the metric for comparison.

Our traces were collected from multiple sources, including the internal and external Intel web servers, and connections between university networks and the internet core, obtained from the National Laboratory for Applied Networking Research (NLANR). The NLANR traces are no longer available on the web, but this paper gives a good overview of their characteristics [29]. The ANL traces are collected from the link between the Argonne National Lab and its internet service provider. This is an OC-3 (155 Mbps) link and each trace contains about 0.5 million packets. The MRA traces are from the link connecting Merit and Abilene- two large networks. This is an OC-12 link and each trace contains about 5 million packets. The MRA link is closer to the core of the internet while the ANL traces are at the edge. We believe these to be a good sampling of real world workloads likely to be encountered by Snort. Since these traces only include packet headers, synthetic packet bodies were inserted for testing.

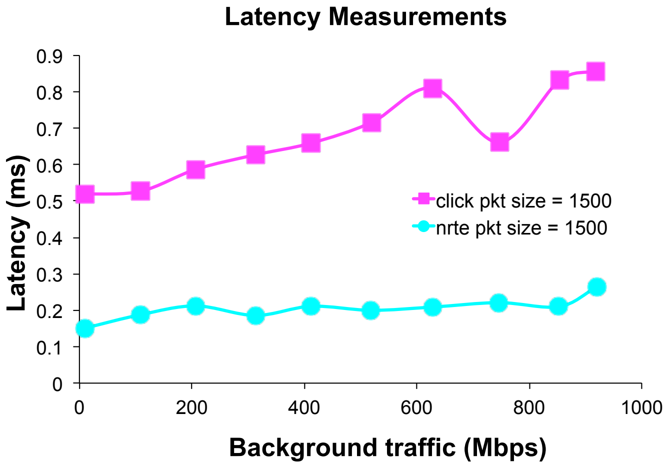


Figure 4.7: Latency Measurements

These bodies contain random data and are not meant to simulate any particular attack signature.

The results are in Figure 4.6. As can be seen from this graph, the multithread enabled RTE-Snort significantly outperforms the single-threaded original Snort in processing all workloads. While the speedup of the ANL traces can be attributed to the parallelism that we were able to extract via pipelining, the big speedups in processing the MRA traces are due to the large number of flows (25000) in these traces. We were able to leverage this parallelism in the data by replicating the TCP stream reassembly stage (the bottleneck stage) and pinning subsets of the flows to each replica. In this fashion, the NRTE allows more hardware resources to be effectively applied to the application bottleneck.

4.4.5 Latency

For our final experiment, we examine the latency of packets traversing our IPv4 router implementations. The experiment is set up the same as that in section 4.4.2, but this time, a ping is sent from one machine connected to the router system on a different port to another machine also connected to the router. This was done with varying levels of background traffic, generated in the same manner as in the previous IPv4 forwarder experiment. The latency for a reply was measured at the ping sender. Figure 4.7 shows the results of this experiment for the Click router and the NRTE router. For both large and minimum sized packets, the NRTE based router produces much lower round trip times than the Click router.

Chapter 5

Scheduler API

We have demonstrated the effectiveness of our NRTE interface for writing networking applications in chapter 4, as well as the shortfalls of simple packet allocation strategies in chapter 3. In this chapter, we examine the use of more complex packet scheduling algorithms and their application to the NRTE. We present an interface for writing new schedulers and demonstrate the different performance profiles achieved using different scheduling algorithms. We start by reintroducing the NRTE application interface and the updates needed to deal with dynamic scheduling. We follow with a description of the scheduler programming interface and a description of the two schedulers we tested.

5.1 NRTE

The Network Runtime Environment (NRTE) is our API for writing efficient multicore networking applications. Specifically, we target applications that can be effectively pipelined but require a large amount of per flow state and guarantees about packet ordering and dropping. For example, searching a packet payload for regular expressions is most effective if it is done over a fully reassembled and in order stream. Dropping packets from within the reassembled stream or processing them out of order could cause the search to produce false negatives.

We provide two sets of mechanisms to achieve our goals. First, the application API provides efficient inter-core communication, flow pinning, and flow control to prevent unwanted dropping of packets once they enter the pipeline. These mechanisms are useful for stateful applications such as intrusion detection which perform better when packets in a single flow arrive in order. It should be noted that while flow information is collected by the API, not all schedulers will support flow pinning.

Our second mechanism is a backend scheduler API. The NRTE is capable of dynamically adjusting the application pipeline to adapt to different environments. This can include running on hardware with varying numbers of cores, or changing traffic patterns. The best algorithm for this adaptation may vary by application, architecture, and other factors. The NRTE provides a well defined API for allowing users to write new scheduling algorithms.

5.1.1 Application Interface

Users of the NRTE are expected to break their applications into pipeline stages. The NRTE provides two types of stages: explicit stages are threads that run in an infinite loop, whereas implicit stages register a function that is run whenever a packet is enqueued on the associated queue. The NRTE runtime is responsible for mapping these stages to the available hardware cores in the system. If a stage is thread safe and can be easily duplicated, the NRTE runtime may duplicate an implicit stage that requires more processor time and split the traffic between the copies. A fuller explanation of the NRTE front end can be found in the previous chapter or here [54]. Each component in the NRTE has associated adaptation functions to initialize and tear down state when a duplicate is created by the scheduler. This is necessary for applications such as TCP reassembly, which keep a lot of per state information, but can keep it thread local and lockless if flows are pinned. Each component also has a classify function associated with it so that flow membership can be determined by the application programmer and not fixed by the API.

In order to accommodate scheduling and dynamic adaptation, the underlying architecture of the NRTE has changed greatly from that presented in the previous chapter. Each core has a single thread pinned to it. All components scheduled to run on that core are run in this thread context. The thread cycles through all the stages scheduled to run on it. Explicit stages are run on each cycle through the stage list. Implicit stages are run if there are any pending packets on their incoming queues.

There is a scheduler thread that takes care of schedule computation. It wakes up every scheduling interval (currently 10 seconds), pauses all the processing threads, reads and resets all the statistics buffers, and reads the updated schedule. Schedule computation itself is allowed to run in parallel to other tasks as some schedulers can take a long time to run. This is achieved by having 2 sets of buffers for the statistics and 2 copies of the schedule. The live version of each is swapped at each scheduling interval. After reading the new schedule, the scheduler thread sends updates to all the processing threads, at which point they wake up and make necessary adjustments (adding and removing components and their associated queues). If there is a backlog of packets in a queue that is being removed from the system, its contents are redistributed to the remaining copies of that stage in a manner that is defined by the scheduler. With careful design of our data structures, we have been able to eliminate almost all locking from normal operations in the NRTE internals.

5.1.2 Scheduler Interface

The Scheduler API provides the interface between the NRTE runtime and the scheduling algorithms that map components to cores. We have written several schedulers that conform to this API, which we describe later in this chapter.

The scheduler API is embodied in the ScheduleBuilder class, which provides a set of virtual functions, listed in table 5.1, that will be called by the NRTE's frontend to calculate new schedules. All schedulers must subclass the ScheduleBuilder and provide instantiations of these functions. With the exception of feeding the Scheduler the logical topology, that is, the configuration of the components and the queues between them, the scheduler interface is not directly accessed by the application programmer. Rather, these functions are used by the NRTE runtime to automatically adjust the physical topology of the system to adapt to new conditions. A full listing of the ScheduleBuilder's member functions is given in appendix B.

All topology information is stored in the ScheduleBuilder. When a programmer creates stages with the NRTE frontend, they are automatically registered with the ScheduleBuilder by the API. Further topology information, such as the logical edges in the system, must be specified by the programmer.

Table 5.1: SchedulerBuilder Class Member Functions

Function Name	Description
addComponent	Register a component with the scheduler
addOutputQueue	Register a queue with the scheduler
pinComponent	Force a component to run on a particular core
finalize	Tell the runtime that all topology information has been input
computeSchedule	Compute a new schedule
selectVirtualQueue	Tells the runtime which copy of a component to send the next packet to
getComponentMapping	Returns a list of cores on which a particular component is run
getEdges	Returns a list of the cores on which a component may send packets

Once the topology has been fed to the scheduler, the NRTE can call the ScheduleBuilder to compute new schedules. Information about the new topology is exported to the NRTE runtime through a series of functions exported by the ScheduleBuilder class. These include the location of physical component instances and the existence of edges between components on different cores.

Information also needs to flow to the ScheduleBuilder from the NRTE. The Statistics class represents an abstract class interface that is implemented by the NRTE runtime to communicate runtime statistics to the ScheduleBuilder. These include the number of packets processed or dropped in each stage, the amount of processor time used by each component and the remaining backlog on each queue. A summary of the Statistics member functions are listed in table 5.2. These are the counters that are relevant to the schedulers we have implemented. Different statistics classes could be created for different schedulers if new algorithms arise that need other information.

Table 5.2: Statistics Class Member Functions

Function Name	Description
getVirtualComponentTimeUsed	Returns the total number of sec of processing time used by this component on the specified core during the previous measurement window.
getVirtualComponentPacketsProcessed	Returns the total number of packets accepted and output by the given component on the given core during the last measurement window.
getVirutalComponentEdgeDiscards	Returns the total number of packets discarded by the given component on the given core, destined for the given destination during the last measurement window.
getVirtualComponentEdgeOutput	Returns the total number of packets output by the given component running on the given core to the given component on the given core, on the queue with the index.
getVirtualOutputQueueBacklog	Returns the total number of packets backlogged on the output queue at end of measurement window.
selectVirutalQueue	Tells the runtime which copy of a component to send the next packet to

5.2 Schedulers

One of the major advantages of the scheduling framework is that we can track network traffic and processing characteristics, and adapt to changes. We consider two algorithms which rely on this information to compute multicore schedules adapted to the current traffic.

The first algorithm we consider is the one described in [52]. The authors, working from a Click [30] infrastructure, submit the hypothesis that previous solutions suffer from a lack of task granularity. That is, the system consists of a small number of monolithic components, often of high processing requirements. This makes it difficult to assign instances to cores with any reasonable hope of obtaining a balanced load. They present a new algorithm which decreases task size by duplicating components. These component instances are assigned to cores with the goal of distributing processing load as evenly as possible. For example, a high workload component such as IPSec Decryption could be duplicated three times, with each instance running on a different core, and traffic for the component split evenly among them. In assigning these instances to cores, an attempt is made to keep the components' pipelines of components together on the same core when possible. The major goals of the algorithm are: balance processing load as evenly as possible among cores, and compute schedules very quickly to adapt to changing loads as rapidly as possible. We will refer to this algorithm as the WW algorithm, after the authors' initials.

We now consider a second algorithm designed to overcome some of the WW algorithm's shortcomings. The WW algorithm relies on the implicit assumption that per-packet processing requirements are large compared to the cache impact of moving packets from core to core. For pure forwarding applications such as IPv4, this is not the case. In experimental evaluation of the WW algorithm, it was found that IPv4 packet processing performed better on a single core than balanced across 8 cores. Based on the hypothesis that unnecessary inter-core crossings were the cause of poor performance, we created a new scheduling algorithm based on two new goals: do not overload any cores, and minimize inter-core crossings. This algorithm relies on the additional functionality that components need not be split by integral duplications, and that traffic from a given instance need not be split evenly among downstream component instances. The component graph schedule is formulated as a linear program, with constraints that ensure that no individual cores are overloaded. The objective of the linear program is to minimize the number of packets sent from core to core. In consequence, we also tend to minimize the number of cores in use. We refer to this as the LP algorithm.

Our implementation of the WW algorithm has a minor change from the original algorithm as published. The WW algorithm was used to create integral numbers of tasks, and each task corresponded to a call to the component from a thread scheduler. That is, if two copies of a single task were assigned to a given core, the component would be called twice in a scheduling round to process a packet on that core. Each instance of the same component shared the same input queues, so only the thread scheduling enforces the schedule. In our scheduling framework, however, all tasks on the same core become a single aggregated component instance. Each instance has a separate input queue and processes packets as they become available. The schedule is enforced by upstream components, which send packets to instances based on the precomputed schedule.

The LP algorithm has some difficult implementation edge cases. First, the formulation of the linear program propagates measured input loads forward through the component graph to predict inter-core crossings. In edge cases due to measurement granularities, it is possible for the linear program to predict a system overload which allows no feasible solution. In these cases, we simply fall back to the WW algorithm. Because this only happens when per-packet processing is very high, the impact of inter-core crossings is negligible. Second, if a component is never used during a measurement window, the linear program solution need not schedule that component at all. We ensure that all components are scheduled by assigning unused components to a ”dump” core.

Both algorithms are implemented in a shared scheduling system which gathers all statistics from the running system, then calls the scheduling algorithm to compute scheduling weights between component instances. Once the desired scheduling weights between component instances is available, these are used to generate weighted deficit round robin (WDRR) packet schedulers at the egress of each component. During actual packet processing, both scheduling algorithms rely on exactly the same implementation.

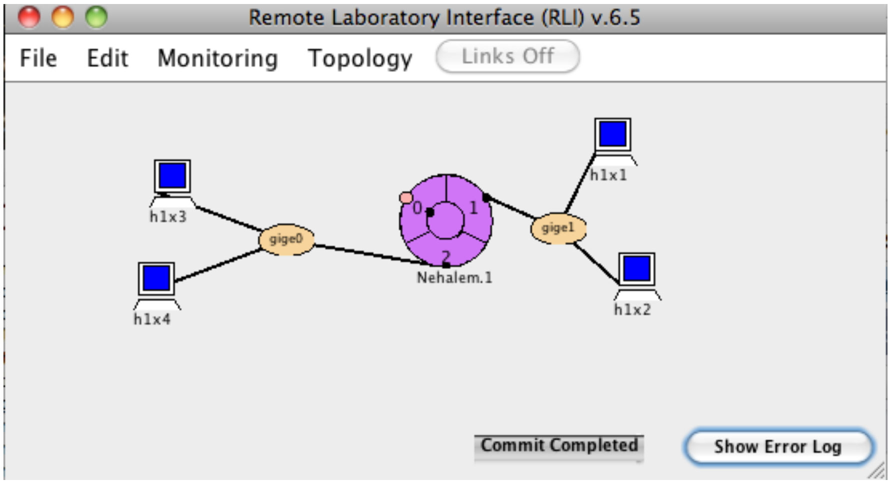


Figure 5.1: ONL Configuration

Schedule changes are done at longer time intervals than with the original WW, as our LP scheduler takes a longer time to run and our components are expected to maintain state that needs to be initialized or torn down on each reschedule.

5.3 Synthetic Benchmark

We created a benchmark to test our API and schedulers based on the IPSec pipeline used in [52]. Our version is not a real IPSec router, but the topology and workload mimics one. We artificially lengthened the pipeline and created a dummy encryption workload based on the blowfish encryption algorithm. This was done in order to create a workload capable of stressing our hardware at 1 Gbps traffic. A diagram of this topology is displayed in Figure 5.2. All traffic goes through the initial pipeline that mimics an IPv4 forwarder performing a series of header checks and other routine bookkeeping. At the demux component, normal IPv4 traffic is sent to the forwarder component, which lookups the next hop and sends traffic out the proper interface. Specially marked traffic enters the encryption loop, where a dummy encryption workload is performed on the packet body and before it is reinjected into the IPv4 pipeline. This mimics the action of an IPSec router decrypting and decapsulating the packet.

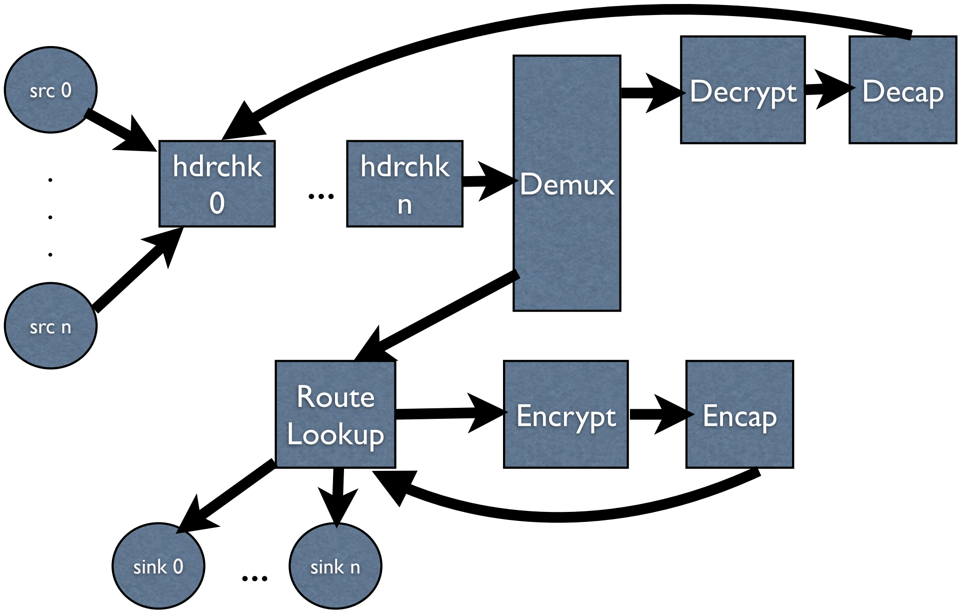


Figure 5.2: Benchmark Logical Topology

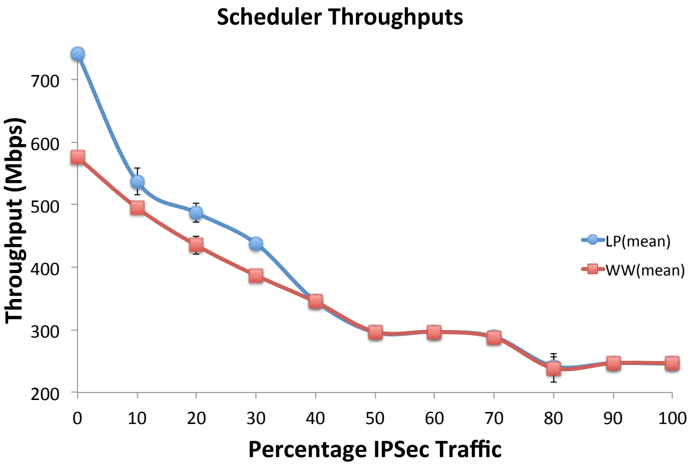


Figure 5.3: Scheduler Benchmark

Our test setup involves sending UDP packets with 1300 byte payloads from one host to another with a test system in the middle acting as a router with our dummy workload. This topology was setup in the Open Network Lab (ONL) [51], a reprogrammable network testbed. The system under test running our pipeline is an 8 core Nehalem with 12 GB of memory. 2 end hosts are connected to each of the NICs on the Nehalem through a gigabit switch. This topology is illustrated in Figure 5.1. Our benchmark consists of sending traffic in one direction and measuring the maximum throughput reached in the steady state by each scheduler as we vary the percentage of the traffic that goes through the IPSec loop. Figure 5.3 shows the results of this experiment using both the WW and LP schedulers. The data used to generate the graph and calculate confidence intervals is given in table 5.3. We can see that the LP performs better at lower levels of IPSec traffic, while the two schedulers have similar performance above 40 percent IPSec traffic. This demonstrates the superiority of the LP under certain conditions and the effectiveness of the NRTE's scheduler interface in allowing us to test different schedulers on the same application program. Given the topology we used, this result is exactly what we expected to see. The results are robust when using a 95 percent confidence interval on the mean using a t-test with 4 degrees of freedom.

Table 5.3: Scheduler Benchmark Data

% IPSec Traffic	LP (Mbps)	var	WW(Mbps)	var
0	741	.014	575	.17
10	537	291.9	495	.0001
20	487	143.5	435	126.07
30	438	5.59	387	.288
40	346	3.7e-5	345	.8
50	296	1.8	297	5.0e-5
60	297	.0009	297	9.0e-5
70	289	1.6	288	.46
80	242	142.2	239	324
90	247	.4	247	.0003
100	246	.49	247	.0007

One result that differs from theory is that the LP takes a longer time to converge to the steady state than the WW. We believe this is due to the fact that we are collecting statistics in user space, which means the schedulers are not being fed data on the number of packets being dropped in the kernel. Nor is the scheduler able to account for the amount of processor time used to process packets. In the future we would like to find a way to account for these resources and see if that allows the LP to converge more quickly, as it should in theory.

5.4 Regex Application

To test our API on a completely functional, real world task, we developed an inline deep packet inspection application. This application consists of 4 stages- packet reception (RCV), TCP reassembly (RSM), regular expression matching (REG), and IP forwarding (FWD). The regular expression matching is based on the hybrid finite automata (HFA) presented by Becchi and Crowley in [8]. To our knowledge, this is the first fully functional deep packet inspection application to incorporate this work. The only live packet testing done before assumed UDP packets with signatures that did not span multiple packets[9]. Packets proceed in the order RCV, RSM, REG, FWD.

The RCV and FWD stages are the simplest. The RCV stage is an rte_explicit stage that continuously reads packets from a socket and enqueues them to the RSM stage. The FWD stage is an implicit stage that reads packets passed to it and forwards them out the proper socket. This involves reading the incoming socket from the packet's metadata, looking up the outgoing device from a hash table, and doing a socket write.

The RSM stage reassembles TCP streams before sending them to the REG stage. This is an important task as the regular expression parsing needs to work over fully reassembled, in order streams in order to properly detect the target signatures. Otherwise, matching signatures that span multiple packets can be missed. The RSM stage passes packets to REG stage as they are ready- in order packets are immediately passed, while out of order packets are buffered until the missing holes are filled, and then the entire backlog is passed on. This piecemeal reassembly is necessary because our application performs all of this inline.

The REG stage performs regular expression matching on the packets passed in from the RSM stage. This stage assumes that no packets are dropped in the handoff from the RSM stage, as proper regex matching requires the fully reassembled stream. This is supported by the transaction mechanism of the NRTE described above. When an enqueue is attempted by the RSM stage to a full queue, it detects this and backs out the state changes and puts the packet back at the head of the incoming queue. Processing will resume later as if that packet had never entered the stage.

The regex matching is performed by treating the body of the incoming packet as an input string to a hybrid FA constructed from a set of regular expressions fed in at application startup. The FA is traversed and the state of the traversal is saved so that it can resume when the next packet of that flow arrives. This allows us to detect regular expressions that span multiple packets.

The pipeline can be further deepened by using multiple REG stages that each search for a different set of regular expressions. This is useful for creating multiple, smaller state machines. If the state machines can be reduced in size to fit within the cache of a processor, we expect to realize some performance gain.

5.4.1 Evaluation

We evaluated the regex matching application by testing it against both live traffic and by feeding it artificial packet traces. The regex tool provided by [8] includes a tool for generating streams that match a ruleset with a certain probability. We generated streams that matched at different probabilities for a variety of rulesets.

The network configuration tested is shown in figure 5.1. The system that runs the regex application serves as a router, with senders and receivers connected to different ports so that all traffic must go through it.

We found that throughput was hindered by the locking required to synchronize access to shared flow specific data structures. We tested different static mappings by manually configuring them. When we map a single copy of each stage to its own CPU, we get a baseline of 50 Mbps. This drops to 20 Mbps when we allow the regex stage to be duplicated on 5 processors by a mesh scheduler that puts copies of all nonpinned stages to all cores. We believe this is due to lock contention. If we modify the pipeline to dispense with reordering packets, the same topology achieves 250 Mbps, which is consistent with the linear speedup expected when duplicating the bottleneck stage from 1 to 5 cores. This is consistent with experiments we performed in chapter 3.

The lesson we take away from this set of experiments is that we need to develop a scheduler that will schedule flows rather than packets. Such a flow pinning scheduler would allow all packets in a single flow to be processed only in a single copy of each stage. In this scenario, we can eliminate locks on per flow data structures. In the regex application, this would allow us to eliminate all the locks. We can also eliminate checks for packet reordering in the REG stage. When flow-pinning, both the REG and RSM stages can keep local per thread copies of flow state that will not be accessed by other threads, as packets for that flow will only be processed on that copy of the stage. This eliminates the need to access and modify the global state table for every incoming packet, which is inefficient and requires locking. Access to the global table is only needed when a flow is accessed on a core for the first time, or when a flow is migrating to a different core when the mappings change. In the first case, the state is copied into the thread local cache, and in the latter it is written back to the global table.

We believe a flow pinning scheduler can be achieved with a minor modification of the LP scheduler, but more substantial engineering effort would be required to modify the NRTE runtime to pass flow liveness information to the scheduler. This is left for future work.

5.5 Related Work

Deep packet inspection is performed by many intrusion detection systems, including Snort [42], Bro [38] and PAM. These applications use a different regex engine and take a different approach to parallelization. Attempts to parallelize these applications include [34] [6] [27] [45] [7] [46]. None of these use either the regex engine or the scheduling algorithms we present in this paper. Furthermore, none of them are built atop an API that allows their insights to be reused in other applications.

Click [30] and its derivatives provide a framework for building software routers by breaking them into small components and connecting them using a high level language. Click targets a somewhat different niche. Click elements usually represent a small simple computation, and it can be difficult to write complex, stateful applications in this manner. Furthermore, on multicore systems, Click uses a work stealing scheduler to allocate work to processors. We provide multiple schedulers and a documented API to create new ones. There have been many other proposed APIs [44], but none of them provide the scheduler flexibility that we do.

5.6 Conclusion

We have presented a dynamically adapting framework for writing complex, stateful network processing applications with an additional API for writing new schedulers. We demonstrated the usefulness of the API in achieving different performance characteristics under different scenarios using different schedulers. We also used our API to write a deep packet inspection application that demonstrates the usability of our API for real world applications as well as pointing the way to future refinements.

Chapter 6

Conclusion

6.1 Summary

Multi-core architectures hold great promise for making high performance networking applications that are easy to create and modify. Different strategies for exploiting this parallelism require modifications to current generations of hardware and software.

In this dissertation, we have examined both hardware and software methods for exploiting the parallelism of modern multi-core CPUs. On the hardware organization side, we have demonstrated the usefulness of network onloading using a cluster of specialized cores. On the software side, we have designed an API for writing pipelined networking applications and demonstrated its usefulness in parallelizing existing applications. We also showed the usefulness of a scheduler API for creating new algorithms to map the logical pipeline to the underlying hardware. The efficacy of different scheduler algorithms under different circumstances has also been shown.

6.2 Future Directions

Network onloading can be improved by exploiting new hardware mechanisms such as I/O MMUs that have become available in some new architectures. As discussed in chapter 2, forcing the host processor to handle page table pinning and virtual address translation for the onload engine is a major overhead.

Other things to investigate include the effect of onloading in a system with more general purpose cores. Our prototype used a single onloader to service a single general purpose core. The proper organization of a more complex architecture is an open area of research. There are issues such as synchronizing access to the onloader and the number of onloading engines needed to provide enough bandwidth for multiple cores.

There are multiple directions for future research regarding the NRTE. The first is the development of a flow-pinning scheduler. This would allow us to support applications such as the deep packet inspection engine from chapter 5. The investigation of the interplay between the Linear Programming and Wolf and Wu schedulers under realistic workloads will be very interesting.

Appendix A

NRTE API

A.1 Terminology

Pipeline A user created graph connecting various application stages. There are no restrictions on the kind of graphs that might result from these interconnections. For instance graphs with cycles or with an arbitrary degree of fan-out are considered to be valid pipelines.

Stage A logical entity that performs some processing on elements as they pass through a pipeline. A stage typically works on an element, then passes it on to the next stage in the pipeline, and then goes back to get the next element to work on from its upstream stage. There are two kinds of stages (Explicit and Implicit input) depending on their style of receiving the elements they process.

Explicit Input Stage A pipeline stage that obtains the elements that it processes by explicitly dequeuing the elements from a queue or reading from a NIC. Thus, in addition to the code for element processing an explicit input stage will have code to extract elements from its source.

Implicit Input Stage A pipeline stage that processes elements that are passed to it as input parameters. The code in the stage is only concerned with processing the element it receives. It terminates after processing the given element. As such it relies on another entity to remove elements produced by the upstream stage in the pipeline.

Stage Instance When a stage is actually executed in a thread context, it is referred to as a stage instance.

Flow A concept whereby various elements of a stream of data are related to each other in an application defined manner. A TCP connection is an example of a flow in which all the packets of the connection have the same 5-tuple (source IP address, destination IP address, source port, destination port, protocol). Typically an application that is processing a flow will access some flow state for processing each element of the flow.

Flow Pinning An optimization technique wherein all the elements of a flow are sent to the same core. This improves cache utilization since all the processing for the elements of the flow that requires access to the flow state happens on the same core. Without flow pinning the elements are sent to different cores resulting in the flow state having to shuttle back and forth between the caches of the cores which is inefficient.

A.2 Common Data Types

rte_queue_id_t
Definition typedef uint64_t rte_queue_id_t
Description Type used to specify queue identifiers

rte_flow_classifier_func_t
Definition typedef unsigned int (*rte_flow_classifier_func_t) (uint64_t elem)
Description Pointer to function that identifies the flow of a given element. This function returns a flow identifier.

rte_explicit_input_func_t
Definition typedef int (*rte_explicit_input_func_t) (void *cookie)

Description Pointer to function that does the processing associated with an explicit input stage. If an explicit input function is registered (refer to the rte_register_explicit_input_function function), the RTE creates a thread of execution for the function when the RTE starts. The RTE does not duplicate these stages (hence the function need not be thread-safe). However the RTE decides where to execute an explicit input function and might choose to change the mapping at run time.

Any state that the function might need during execution (like the queue(s) to read from) can be passed in using the argument it accepts. The RTE is given this information when the function is registered.

The explicit function is essentially the body of an infinite loop, but with control and invocation left to the runtime. It is meant to be used as a packet source. The return value should be one of the RTE return codes. See the section on return codes for a description.

rte_implicit_input_func_t

Definition typedef int (*rte_implicit_input_func_t) (uint64_t stage_id, unsigned int flow_id, uint64_t elem)

Description Pointer to function that processes the given element in the given flow. This type is used for stages that do not explicitly read from a queue. The stage is associated with a queue (refer to the rte_register_queue function) and the RTE assumes the responsibility for dequeuing elements from the queue and invoking the stage; i.e., the function pointed to by this function pointer. In addition the RTE identifies the flow the dequeued element belongs to and passes in the flow identifier along with the element when this function is invoked. The stage_id is the value that was registered with the RTE when the queue was created. It may be used to disambiguate between multiple queues that use the same function.

At the time of queue creation the programmer can specify whether this function can be duplicated. If duplication is allowed, this function MUST be thread-safe since new threads could be created to execute this function at the discretion of the RTE.

Note that this function is expected to return after processing the given element. It will be invoked by the RTE for every element read off the queue with which this function is associated. The return value should be one of the RTE return codes described in another section.

rte_adapt_callback_func_t

Definition typedef void (*rte_adapt_callback_func_t) (uint64_t stage_id, rte_cb_status, status, unsigned int num_instances)

Description Pointer to function that is invoked by the RTE when the stage associated with the given queue is adapted. The association between the stage and the callback function is made when the queue which feeds the stage is created (refer to the rte_register_queue function). This function is intended to be used by stages that modify their internal state based on the number of copies of the stage. The stage_id is the value that was registered with the RTE when the queue was created. It can be used to disambiguate between multiple queues that use the same adapt callback function. The status can be RTE_CB_SETUP, RTE_CB_NUM_CHANGE or RTE_CB_DEAD. Setup tells the function that a new duplicate of the stage is being created in this thread, allowing per copy initialization to be performed. Dead tells the function that the associated copy of the stage is going away, and num change tells the function that an existing copy is going to continue running but the number of copies in the system has changed.

A.3 Initialization and Shutdown API

rte_start

Signature *int rte_start(void)*

Description Starts the run time environment. Prior to this call, all necessary application components must be registered with the run time (refer to the rte_register_explicit_input_function and rte_register_queue functions). Once started, the RTE performs any internal initializations and then creates new threads to start executing the registered application components.

Paramaters In None

Paramaters Out Return value <= 0 indicates failure; >0 indicates success

rte_stop

Signature *int rte_stop(void)*

Description Stops the run time environment. Once this function is invoked, no new data is read in by the application. Once all the data that is already in the application pipeline is processed, the threads created by the RTE are terminated.

After the RTE is shutdown the rte_start call returns allowing the main application thread to perform any application state cleanup.

Parameters In None

Parameters Out Return value <= 0 indicates failure; >0 indicates success

rte_register_explicit_input_function

Signature *int rte_register_explicit_input_function(rte_explicit_input_func_t explicit_stage, void *cookie)*

Description Creates a thread of control to execute function explicit_stage. The RTE will not attempt to duplicate this thread; however it can choose to map it to any core on the system.

Parameters In *explicit_stage* Pointer to function to be registered with the RTE

cookie Parameter passed to the registered function when it is invoked

Parameters Out Return value <= 0 indicates failure; >0 indicates success

rte_register_queue

Signature *rte_queue_id_t rte_register_queue(rte_implicit_input_func_t implicit_stage, uint64_t stage_id, boolean allow_duplication, rte_flow_classifier_func_t flow_classifier, boolean always_classify, rte_adapt_callback_func_t adapt_cb)*

Description Creates a queue and returns a queue identifier that can be used to refer to the created queue.

Parameters In *implicit_stage* Pointer to function to be called to handle the element retrieved from the queue.

A NULL value can be passed to instruct the RTE to not retrieve from the queue. In this case it is the programmers responsibility to remove and process the elements from the queue. When implicit_stage is NULL, all the other parameters are ignored.

stage_id A user specified nonce value that will be returned by the RTE in any callback associated with this queue. This is useful when the same function is used to handle multiple queues. The stage_id value can be used to identify the context in which the specific callback should be processed. This value is opaque to the RTE and pointers to the stage context can be passed in as well.

allow_duplication Flag indicating whether the RTE is allowed to duplicate (i.e. create new threads to run) the stage pointed to by implicit_stage. Typically this flag should be set to true; it is provided to include in a pipeline legacy code that is not thread safe.

flow_classifier Pointer to function that classifies the elements retrieved from the queue into its constituent flow. The RTE calls this function on each element added to a queue. If the RTE duplicates the receiving stage, the flow identifier is used (flow id modulo num instances) to determine which copy receives the element. This ensures that elements belonging to a flow, as identified by this function, are sent to the same copy of the implicit stage. This prevents any cached flow state from having to migrate between different threads. This also avoids reordering of elements in a flow.

If no flow classifier function is specified (flow_classifier = NULL) an element can be sent to any stage instance at the choice of the RTE.

Since the RTE uses the flow classifier function to determine an elements flow identifier, the RTEs ability to distribute the flows amongst the copies of the implicit stage depends on the effectiveness of this function. Also the ability of the RTE to maintain element order within a flow depends on this function. For instance, if this function changes its definition of a flow over time the RTE cannot provide any guarantees. Finally, if stricter definitions of element ordering need to be imposed it is left to the application developer to enforce these (for instance by adding an extra stage that manages ordering)

always_classify The RTE need not invoke the flow classifier function when there is only one copy of the receiving stage. In this case the flow id passed in to the receive stage will be 0. However, if the receive stage uses the flow id for its processing it can request the RTE to classify even when there is only one copy of the stage. This can be done by setting the always_classify flag to true.

adapt_cb Pointer to callback function that is invoked whenever the RTE changes the number of copies of the stage.

Parameters Out rte_queue_id Returns a queue identifier referring to the created queue, 0 if the queue could not be created.

A.4 Queuing API

The queuing API supports dequeuing and enqueuing of elements from queues. These functions allow a developer to break an application into a pipeline while passing information into the RTE that allows: identification of the parallelizable components, monitoring of the pipeline stage load, adaptation of the pipeline configuration.

These functions aim to minimize the impact of adding queues into an existing application (by automatically dequeuing elements for implicit stages) while allowing for applications where the developer might require more control over how the elements of different queues are handled.

rte_enqueue

Signature *int rte_enqueue (rte_queue_id_t qid, uint64_t elem)*

Description Adds the element elem onto the queue referred to by qid. While any 64 bit value can be enqueued, it is expected that this will typically be a pointer to some data (typically packet data). When a pointer is enqueued, the sender must ensure that data pointed to by elem is around when a receiver looks at it later. This function blocks until elem is successfully placed on the queue.

Parameters In *qid* Identifier referring to the queue on which data must be added.

elem The data to enqueue

Parameters Out Return Value <= 0 indicates error, the caller retains ownership of elem in this case; >0 indicates success

Appendix B

Schedule Builder API

The ScheduleBuilder has a series of life stages through which it proceeds under the control of the caller.

B.1 Phase 0 pre-instantiation

A *ScheduleBuilder* (or subclass) is instantiated by the construction:

ScheduleBuilder(unsigned int numProc, unsigned int cacheMax)

numProc is the number of cores for which well schedule. This parameter MUST be a positive integer, or *sbException* will be thrown.

cacheMax is an optional parameter for caching schedulers. If unspecified, the scheduler does not cache. If specified, it is the number of pre-computed schedules to keep around for faster lookup. Not currently implemented but may be a future possibility.

After construction, a *ScheduleBuilder* is in Phase 1, Topology Creation.

B.2 Phase 1- Topology Creation

At this point, the *ScheduleBuilder* has no knowledge of the directed component multigraph. No schedule may be computed, and attempts to use the scheduling methods will throw *sbException*.

Multithreading note: the Topology Creation methods are not guaranteed to be thread safe.

Components are referenced by handles of type *handle_t*. *handle_t* is an alias for *(void*)*, but obviously any bitstring of appropriate length may be used.

Components can be added to the *ScheduleBuilder* in two ways: explicitly or implicitly.

Explicit Component Addition

addComponent(handle_t componentHandle, boolean isSource, std::string name)

componentHandle is treated as an opaque bit string of length sizeof(void*). It MUST be unique (never been added previously) or *sbException* will be thrown. Either *isSource* or name may be omitted, and these parameters may appear in any order. The optional *isSource* flag designates the component as a source of packets; that is, packets from the network may originate here. If not specified, *isSource* defaults to false. The name is an optional parameter used by any output methods (primarily debugging methods) to identify the component in output. If not provided, the component will be identified by an assigned number. The name need not be unique (components may have identical names).

Implicit component addition happens when an unknown component handle is referenced in any topology setup method, such as by adding a queue to/from it or changing the optional attributes. Implicitly added components are not sources, and have assigned *names*.

Adding Output Queues

Queues between components are added using *addOutputQueue*. If a component not previously seen is used as a queue endpoint, the component is implicitly added to the topology.

addOutputQueue(handle_t tailComponentHandle, handle_t headComponentHandle, unsigned int index)

tailComponentHandle is the handle of the *writer* to the Queue; that is, the tail of the edge. *headComponentHandle* is the handle of the *reader* of the Queue; that is, the head of the edge. The *index* is an optional unique identifier for disambiguating multiple queues between the same components. For efficiency, it is strongly recommended that they be contiguous numbers starting from 0. (The vertex implementation may allocate an array from 0 .. maxEdgeIndex.) If unspecified, the index defaults to 0, the normal case when the system is not a multigraph.

Optional Attributes

Components may be given optional attributes to control how they are scheduled.

Marking components as non-parallelizable

Most multi-core schedulers anticipate being able to schedule the same component on multiple cores as once. Not all components can be safely parallelized. A component may be marked in the topology as non-parallelizable, and the scheduler will not schedule that component on multiple cores regardless of consequences. This may lead to violating core capacity constraints for schedulers which consider this factor.

void setComponentNoParallel(handle_t componentHandle)

As with other methods which accept a component handle, unknown components will be implicitly added to the topology.

Pinning a component to a core

A component may be pinned to a specific core. That is, the component must be scheduled on exactly and only that core regardless of consequences. This may lead to violating core capacity constraints for schedulers which consider this factor. The component is considered implicitly non-parallelizable.

void pinComponent(handle_t componentHandle, int core)

As with other methods which accept a component handle, unknown components will be implicitly added to the topology.

There is no provision for pinning a component to a set of cores.

Finalizing

Once a Topology has been fully entered, it must be finalized before schedules may be computed. Finalization also validates the topology for any errors. If the topology is found to be invalid, sbException is thrown. It is possible that different ScheduleBuilder subclasses may validate differently, and that the same topology may be valid under one subclass but invalid under another. For example, the TrivialScheduler will permit a NULL Statistics pointer in *finalize()*.

Topologies are finalized as follows, entering phase 2, scheduling.

*void finalize(Statistics *stats)*

The stats pointer is an object which can provides the needed statistics gathering methods. It may acceptably be *NULL* for some schedulers. If the topology is found to be invalid for some reason, *sbException* is thrown. Otherwise, the *ScheduleBuilder* enters phase 2.

B.3 Phase 2- Scheduling

At this point, the topology is locked and no more changes may be made. Further attempts to call the methods of Phase 1 will result in *sbException* being thrown.

Runtime statistics are gathered in measurement windows with an arbitrary length. When statistics gathering begins, the *ScheduleBuilder* must be notified by calling *setMeasurementWindowStart()*.

*void setMeasurementWindowStart(struct timeval *tv)*

tv is an optional parameter used to tell the *ScheduleBuilder* that the current measurement window actually started at a specific time, and is expected to be in the format used by *gettimeofday()*. If not provided or if *NULL*, the *ScheduleBuilder* will use the current time (from *gettimeofday()*).

When measurement windows change, the *ScheduleBuilder* should be notified:

*void flipMeasurementWindow(struct timeval *tv)*

This tells the *ScheduleBuilder* that the current open measurement window has ended, and to begin a new one.

To compute a schedule, call the following method:

void computeSchedule()

computeSchedule() is not reentrant, and multiple calls are not allowed. (Subsequent calls will throw *sbException* until the first call completes.)

The *ScheduleBuilder* will gather all needed statistics from the last (closed) measurement window and compute a new schedule.

To use the scheduler on a per-packet basis, components with packets to send on a queue should call

int selectVirtualQueue(int tailProc, handle_t tailHandle, handle_t headHandle, int index)

The *tailProc* identifies the core on which the component is currently running. The *tailHandle* identifies the component with a packet to send. *headHandle* identifies the destination component. The *index* is optional and identifies the output queue for multigraphs. (If unspecified, index defaults to zero, the norm for non-multigraphs.)

selectVirtualQueue is thread-safe, although there will be a mutex-based performance penalty on simultaneous calls with identical parameters.

Topology Extraction

As convenience functions, the running system can get information about the current schedule.

cpumask_t is an alias for *unsigned long int*.

cpumask_t getComponentMapping(handle_t componentHandle)

Returns a bitmask indicating which processors this component may run on for this scheduling period. For example, bit 0 (the least significant bit) is set to 1 if this component may run on processor 0.

cpumask_t getEdges(int tailProc, handle_t tailHandle, handle_t headHandle, unsigned int index)

Returns a bitmask of the processors to which this virtual component may send on the given logical output queues.

References

[1] Intel Pentium 4 Processor Family Product Information. http://www.intel.com/products/desktop/processors/pentium4, 2008.

[2] RMI: XLR Family of Thread Processors. http://www.razamicro.com/products/xlr.htm, 2008.

[3] Intel quickassist technology. http://www.intel.com/content/www/us/en/io/quickassist-technology/quickassist-technology-developer.html, 2011.

[4] AMD. AMD Opteron Processor Product Data Sheet. http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_739_9003,00.html, 2008.

[5] Boon S. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing on to an i960rn-based iNIC. Hewlett-Packard Technical Report HPL-2001-8, 2001.

[6] S. Antonatos, K. G. Anagnostakis, E. P. Markatos, and M. Polychronakis. Performance analysis of content matching intrusion detection systems. In *Proceedings of the International Symposium on Applications and the Internet (SAINT2004*, page 4, 2004.

[7] Michael Attig and John Lockwood. SIFT: Snort Intrusion Filter for TCP. In *IEEE Symposium on High Performance Interconnnects (HOT-Interconnects)*, 2005.

[8] Michela Becchi and Patrick Crowley. An Improved Algorithm to Accelerate Regular Expression Evaulation. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems 2007 (ANCS 2007)*, December 2007.

[9] Michela Becchi, Charlie Wiseman, and Patrick Crowley. Evaluating Matching Engines on Network and General Purpose Processors. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2009.

[10] Nathan L. Binkert, Lisa R. Hsu, Ali G. Saidi, Ronald G. Dreslinski, Andrew L. Schultz, and Steven K. Reinhardt. Performance analysis of system overheads in tcp/ip workloads. In *Proc. 14th Ann. Intl Conf. on Parallel Architectures and Compilation Techniques*, pages 218–228, 2005.

[11] Jeff Chase, Andrew Gallatin, and Ken Yocum. End-system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39:68–74, 2000.

[12] 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. In *PLDI 05*, pages 224–236. ACM Press, 2005.

[13] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: An overlay testbed for broad-coverage services. *ACM Computer Communications Review*, 33, 2003.

[14] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27:23–29, 1989.

[15] Intel Corp. Intel quad-core Xeon 5300 and Intel 5000p chipset data sheets. http://www.intel.com/design/intarch/quadcorexeon/5300, http://www.intel.com/products/chipsets/5000p, 2008.

[16] Jinquan Dai, Bo Huang, Long Li, and Luddy Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language design and Implementation*, 2005.

[17] Loris Degioanni, Mario Baldi, Diego Buffa, Fulvio Risso, Federico Stirano, and Gianluca Varenni. Network Virtual Machine (NetVM): A new architecture for efficient and portable packet processing. In *Portable Packet Processing Applications, Proc. of 8th International Conference on Telecommunications (ConTEL 2005*, pages 153–168, 2005.

[18] John Dehart, Fred Kuhns, Jyoti Parwatikar, John Turner, Charlie Wiseman, and Ken Wong. The open network laboratory. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, 2006.

[19] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: Exploiting parallelism to scale software routers. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[20] Will Eatherton. The push of network processing to the top of the pyramid. ANCS 2005 keynote address, http://www.intel.com/products/desktop/processors/pentium4, 2005.

[21] Cristian Estan, George Varghese, and Mike Fisk. Bitmap algorithms for counting active flows on high speed links. In *Internet Measurement Conference*, pages 153–166, 2003.

[22] Matt Adiletta et al. The Next Generation of Intel IXP Network Processors. *Intel Technology Journal*, 6(3):6–18, August 2002.

[23] Annie P. Foong, Thomas R. Huff, Herbert H. Hum, Jaidev P. Patwardhan, and Greg J. Regnier. Tcp performance re-visited. In *IEEE International Symposium on Performance of Systems and Software*, pages 70–79, 2003.

[24] Umesh Gajanan. An 8-core, 64-thread, 64-bit, power efficient sparc soc. http://www.opensparc.net/pubs/preszo/07/n2isscc.pdf, 2008.

[25] John Giacomoni, Douglas C. Sicker, John K. Bennett, Manish Vachharajani, Antonio Carzaniga, and Alexander L. Wolf. Frame shared memory: Line-rate networking on commodity hardware. In *Proceedings of the 3rd ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2007.

[26] Pankaj Gupta, Steven Lin, and Nick Mckeown. Routing lookups in hardware at memory access speeds. In *IEEE Infocom*, pages 1240–1247, 1998.

[27] B. Haagdorens, T. Vermeiren, and M. Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *International Workshop on Information Security Applications*, 2004.

[28] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible IP router software. In *Proceedings of teh 2nd USENIX Symposium on Networked Systems (NSDI)*, 2005.

[29] Marina Fomenkov Ken, Ken Keys, David Moore, and K Claffy. Longitudinal study of internet traffic in 1998-2003. In *Proceedings of the Winter International Synposium on Information and Communication Technologies*, pages 1–6, 2003.

[30] Eddie Kohler. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[31] Srihari Makineni and Ravi Iyer. Measurement-based analysis of TCP/IP processing requirements. In *Proceedings of the 10th International Conference on High Performance Computing (HiPC 2003)*, 2003.

[32] Evangelos P. Markatos. Speeding up TCP/IP: Faster processors are not enough. In *Proceedings of the 21st IEEE International Performance, Computing, and Communication Conference (IPCCC)*, 2002.

[33] Jeff Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, may 2003.

[34] T. Nelms and M. Ahamad. Packet Scheduling for Deep Packet Inspection on Multi-Core Architectures. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems 2010 (ANCS 2010)*, October 2010.

[35] Cavium Networks. Cavium networks. octeon cn83xx/cn36xx multi-core mips64 based soc processors. http://www.cavium.com/OCTEON_CN38XX_CN36XX.html, 1998.

[36] Juniper Networks. Open IP solution development program. http://www.juniper.net/partners/osdp.html.

[37] NLANR/DAST. Iperf: The TCP/UDP bandwidth measurement tool. http://dast.nlanr.net/Projects/Iperf/, 2003.

[38] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23-24):2435–2463, December 1999.

[39] Arun Raghunath, Aaron Kunze, and Erik J. Johnson. Framework for supporting multi-service edge packet processing on network processors. In *Proceedings of the ACM First Symposium on Architectures for Networking and Communications Systems*. ACM Press, 2005.

[40] Greg Regnier, Srihari Makineni, Ramesh Illikkal, Ravi Iyer, Dave Minturn, Ram Hugghalli, Don Newell, Linda Cline, and Annie Foong. TCP onloading for data center servers. *IEEE Computer*, 37(11):48–58, November 2004.

[41] Greg Regnier, Dave Minturn, G. McAlping, V. Saletore, and Annie Foong. ETA: Experience with an Intel Xeon processor as a packet processing engine. *IEEE Micro*, pages 23–31, Jan/Feb 2004.

[42] Martin Roesch and Stanford Telecommunications. Snort - lightweight intrusion detection for networks. In *13th Systems Administration Conferece (LISA)*, pages 229–238, 1999.

[43] P. Sarkar, S. Uttamchandani, and K. Voruganti. Storage over IP: When does hardware support help? In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, April 2003.

[44] Douglas C. Schmidt. The adaptive communication environment: An object-oriented network programming toolkit for developing communication software, 1993.

[45] Derek L. Schuff, Yung Ryn Choe, and Vijay S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proceedings of the 12th ACM SIGPLAN PPoPP*, 2007.

[46] Derek L. Schuff and Vijay S. Pai. Design alternatives for a high-performance self-securing ethernet network interface. Technical report, Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium, 2007.

[47] Niraj Shah, William Plishker, and Kurt Keutzer. NP-Click: A programming model for the Intel IXP1200. In *2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Architecture (HPCA-9*, pages 100–111. Morgan Kaufmann, 2003.

[48] P. Shivam and J. Chase. On the elusive benefits of protocol offload. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*, august 2003.

[49] Ion Stoica, Daniel Adkins, Shelley Zhuang, Scott Shenker, and Sonesh Surana. Internet indirection infrastructure. In *Proceedings of ACM SIGCOMM*, pages 73–86, 2002.

[50] Shobha Venkataraman, Dawn Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *Proceedings of Network and Distributed System Security Symposium (NDSS*, pages 149–166, 2005.

[51] Charlie Wiseman, Jonathan Turner, Michela Becchi, Patrick Crowley, John Dehart, Mart Haitjema, Shakir James, Fred Kuhns, Jing Lu, Jyoti Parwatikar, Ritun Patney, Michael Wilson, Ken Wong, and David Zar. A remotely accessible network processor-based router for network experimentation. In *Proceedings of the Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2008.

[52] Qiang Wu and Tilman Wolf. On runtime management in multi-core packet processing systems. In *Proc. of ACM/IEEE Symposium on Architectures for Networking and Communication Systems (ANCS)*, San Jose, CA, November 2008.

[53] Ben Wun and Patrick Crowley. Network I/O acceleration in heterogeneous multicore processors. In *Proceedings of teh 14th Annual Symposium on High Performance Interconnects (HOT Interconnects)*, 2006.

[54] Ben Wun, Patrick Crowley, and Arun Raghunath. Design of a portable router framework. In *Proceedings of the 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, November 2008.

[55] Ben Wun, Patrick Crowley, and Arun Raghunath. Parallelization of snort on a multi-core platform. In *Proceedings of the 2009 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, November 2009.

Networking in the Multi-Core Era, Wun, Ph.D. 2011