# HIGH CAPACITY NETWORK LINK EMULATION USING NETWORK PROCESSORS

by

Abhijeet A. Joglekar

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2004

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Abhijeet A. Joglekar

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Jay Lepreau

John Carter

John Regehr

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

# FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Abhijeet A. Joglekar in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Jay Lepreau
Chair: Supervisory Committee

Approved for the Major Department

Christopher R. Johnson
Chair/Director

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

## ABSTRACT

Network link emulation constitutes an important part of network emulation, wherein links in the topology are emulated to subject the network traffic to different bandwidths, latencies, packet loss distributions, and queuing models. Increasingly, experimenters are creating topologies with substantial emulation bandwidths; contributed both by a large number of low-speed links and a small number of high-speed links. It is a significant challenge for a link emulator to meet this requirement in real time. Existing solutions for link emulation use general-purpose PC-class machines; the well-understood hardware and software PC platform make it attractive for quick implementation and easy deployment. A PC architecture is largely optimized for compute bound applications with large amounts of exploitable instruction-level parallelism (ILP) and good memory reference locality. Networking applications, on the other hand, have little ILP and instead exhibit a coarser packet-level parallelism.

In this thesis, we propose using network processors for building high capacity link emulators. Network processors are programmable processors that employ a multithreaded, multiprocessor architecture to exploit packet-level parallelism, and have instruction sets and hardware support geared towards efficient implementation of common networking tasks. To evaluate our proposal, we have designed and implemented a link emulator, LinkEM, on the IXP1200 network processor. We present the design and a mapping of LinkEM's tasks across the multiple microengines and hardware threads of the IXP1200. We also give a detailed evaluation of LinkEM, which includes validating its emulation accuracy, and measuring its emulation throughput and link multiplexing capacity. Our evaluation shows that LinkEM has a factor of between 1.6 and 4.6 higher throughput for small packets, and link multiplexing capacity between 1.4 and 2.6 higher for low bandwidth links than Dummynet, a popular PC based link emulator, on a comparable PC platform.

To aai-baba.

# CONTENTS

**ABSTRACT** . . . . . . . . . . **iv**

**LIST OF FIGURES** . . . . . . . . . . **viii**

**LIST OF TABLES** . . . . . . . . . . **x**

**ACKNOWLEDGMENTS** . . . . . . . . . . **xi**

**CHAPTERS**

**1. INTRODUCTION** . . . . . . . . . . **1**

1.1 PC-based Link Emulation . . . . . . . . . . 1
1.2 Network Processors for Link Emulation . . . . . . . . . . 2
1.2.1 An optimized architecture for packet processing . . . . . . . . . . 2
1.3 Thesis Overview and Roadmap . . . . . . . . . . 4

**2. BACKGROUND** . . . . . . . . . . **6**

2.1 Emulab . . . . . . . . . . 6
2.2 Link Emulation . . . . . . . . . . 8
2.2.1 Aspects of link emulation . . . . . . . . . . 8
2.3 Dummynet . . . . . . . . . . 10
2.3.1 Emulation model . . . . . . . . . . 10
2.3.2 Implementation . . . . . . . . . . 11

**3. IXP1200 ARCHITECTURE** . . . . . . . . . . **13**

3.1 Hardware Architecture . . . . . . . . . . 13
3.1.1 Programmable processors . . . . . . . . . . 13
3.1.2 Functional interfaces . . . . . . . . . . 16
3.2 Software Architecture . . . . . . . . . . 17
3.2.1 Development platform . . . . . . . . . . 17
3.2.2 Programming model . . . . . . . . . . 18

**4. DESIGN AND IMPLEMENTATION** . . . . . . . . . . **22**

4.1 Software Design on the IXP1200 . . . . . . . . . . 22
4.1.1 Assigning threads on a single microengine to tasks . . . . . . . . . . 23
4.1.2 Assigning microengines to tasks . . . . . . . . . . 24
4.2 LinkEM Design . . . . . . . . . . 27
4.2.1 LinkEM tasks . . . . . . . . . . 28
4.2.2 Mapping LinkEM tasks to the IXP1200 microengines . . . . . . . . . . 41

**5. EVALUATION** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **48**
5.1 Accuracy Validation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.1.1 Bandwidth emulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
5.1.2 Delay emulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.1.3 Loss rate emulation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.1.4 Summary of accuracy experiments . . . . . . . . . . . . . . . . . . . . 51
5.2 Emulation Throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52
5.2.1 LinkEM throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 53
5.2.2 Dummynet throughput . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55
5.3 Link Multiplexing Capacity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
5.3.1 Experiment setup . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 57
5.3.2 Link multiplexing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

**6. RELATED WORK** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **63**

**7. DISCUSSION AND CONCLUSIONS** . . . . . . . . . . . . . . . . . . . . . . **67**

**APPENDIX: EGRESS TASK DESIGN** . . . . . . . . . . . . . . . . . . . . . . **71**

**REFERENCES** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **80**

## LIST OF FIGURES


1.1 Thread-level parallelism on an IXP1200 microengine . . . 3
2.1 Experimental script in Emulab . . . 7
2.2 Topology mapping in Emulab . . . 7
2.3 Dummynet in bridging mode . . . 11
3.1 Architecture of the IXP1200 network processor . . . 14
3.2 An example MicroACE IP forwarding application . . . 21
4.1 Exploiting multithreading on a microengine . . . 23
4.2 Using multiple microengines:pipelining . . . 25
4.3 Using multiple microengines:parallelism . . . 26
4.4 Ingress microblock . . . 29
4.5 Bridging microblock . . . 31
4.6 Classifier microblock . . . 32
4.7 Queuing microblock . . . 33
4.8 Link queue scheduling microblock . . . 34
4.9 Bandwidth emulation microblock . . . 35
4.10 Delay emulation microblock . . . 37
4.11 Loss rate emulation microblock . . . 38
4.12 Timer microblock . . . 39
4.13 Egress microblock . . . 40
4.14 High-level design of LinkEM on the IXP1200 . . . 41
4.15 Microblock Group 1 . . . 44
4.16 Microblock Group 2 . . . 45
4.17 Microblock Group 3 . . . 46
5.1 Throughput measurement experimental topology . . . 52
5.2 Throughput in percentage of line rate (400 Mbps) . . . 53
5.3 Throughput in percentage of line rate (400 Mbps), no drops in egress . . . 55
5.4 LinkEM and Dummynet throughput in percent line rate (400 Mbps) . . . 56

5.5 LinkEM and Dummynet throughput in percent line rate (200 Mbps) . 57
5.6 Link multiplexing with traffic across all four ports . . . . . . . . . . . . . . . . 59
5.7 Link multiplexing topology with traffic across all four ports . . . . . . . . 60
5.8 LinkEM multiplexing with traffic across only two ports . . . . . . . . . . . . 61
5.9 Link multiplexing topology with traffic across two ports . . . . . . . . . . . 62
A.1 Packet transmission on the IXP1200 . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.2 TFIFO to port mapping . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 74
A.3 TFIFO filling with transmission on all four ports . . . . . . . . . . . . . . . . 75
A.4 TFIFO filling with transmission on only ports 0 and 2 . . . . . . . . . . . . 76
A.5 TFIFO/port/fill thread mapping with two microengines for egress . . 78

# LIST OF TABLES

4.1 Measured SRAM/SDRAM latencies for common transfer sizes (cycles) . . . 42
4.2 Hash unit latencies for computing 48- and 64-bit hashes . . . 42
4.3 Cycle counts for Microblock Group 1 . . . 43
4.4 Cycle counts for Microblock Group 2 . . . 44
5.1 Bandwidth emulation accuracy of LinkEM . . . 49
5.2 Base RTT's for a range of packet sizes . . . 49
5.3 Delay emulation accuracy of LinkEM for configured delay of 5 ms . . . 50
5.4 Delay emulation accuracy of LinkEM for configured delay of 30 ms . . 50
5.5 Delay emulation accuracy of LinkEM for configured delay of 101 ms . 50
5.6 Loss rate emulation accuracy of LinkEM . . . 51

# ACKNOWLEDGMENTS

With this thesis, I have completed my graduate studies, and there are a number of people who I want to thank for making these three years an enjoyable and a memorable experience. First of all, I would like to thank my advisor, Jay Lepreau, for giving me an opportunity to work on this project. Jay's encouragement and patience was the primary reason why I did not give up even when hardware idiosyncrasies made progress in the thesis very difficult. I would like to thank my thesis committee members, John Carter and John Regehr, for their timely and constructive comments on my proposal and thesis drafts. Their encouraging words after the proposal talk were a big morale booster.

Mike Hibler and Robert Ricci helped set up the IXP hardware in Emulab. I am also grateful to them for all their inputs, whenever I got bogged down during the implementation and evaluation phase of my thesis. I would like to thank Parveen Patel and Shashi Guruprasad; hopping into their cubicles and discussing my work helped solve a lot of problems. Special thanks go to Erik Johnson, Aaron Kunze, and Dirk Brandewie from Intel. Their prompt replies to all my emails about the IXP hardware and software platform and numerous technical discussions were invaluable.

Late-night coffee breaks with buddies Hemanth, Mohit, and Sujiet helped get my mind off work and recharge my batteries, so that I could get back to hacking for the rest of the night. Finally, thanks to my best friend Geeta, for her patience, understanding, and encouragement for the duration of this thesis.

# CHAPTER 1

# INTRODUCTION

Network emulation [1, 27, 9, 36, 34] provides an environment where real applications and protocols are subjected to an emulated network consisting of resources such as links, routers and background traffic; each such network resource is either real, emulated, or part real and part emulated. Like simulation [33], it is widely used for experimentation in distributed systems and networks. Link emulation constitutes an important part of network emulation, wherein links in the topology are emulated to subject the network traffic to different bandwidths, latencies, packet loss distributions, and queuing models.

Emulab [36], the Utah network testbed, is a large-scale emulation environment used extensively for distributed systems research all over the world. Increasingly, emulation experiments are involving topologies with substantial emulation bandwidths; contributed both by a large number of low-speed links and a small number of high-speed links. There is a need for high capacity link emulators that can meet this requirement in real time. This thesis proposes using network processors for building these high capacity link emulators. To evaluate our proposal, we have developed a link emulator, LinkEM, on the Intel IXP1200 network processor [15]. In this thesis, we present LinkEM's design, implementation and evaluation.

## 1.1 PC-based Link Emulation

Existing solutions for link emulation use general-purpose PCs; the well-understood hardware and software PC platform make it attractive for quick implementation and easy deployment. Dummynet [30], Hitbox[1], NIST Net [25], and ONE [2] are single-node PC-based link emulators that have been used extensively in the research community. They are typically implemented as kernel modules that are transparently interposed between the kernel networking stack and network interface drivers. Packets belonging to an emulated link are subjected to link characteristics and then passed on to the stack or driver. Since these emulators are a part of the kernel, they avoid expensive user-kernel boundary crossings and extra buffer copying, and can thus be implemented very efficiently to squeeze out the maximum performance from the underlying hardware.

A conventional PC architecture typically relies on two important traits in applications to extract high performance: 1) large amounts of instruction-level parallelism (ILP), which can be exploited by the super-scalar architecture and 2) a high locality in memory access patterns so that increasing memory latencies can be hidden by caches. Indeed, as Moore's law allows higher levels of integration, the PC architecture scales up by increasing clock rates, by adding more on-chip memory and by more aggressive exploitation of ILP, thereby improving performance for applications which demonstrate these traits.

Using a set of benchmarks drawn from networking applications, Crowley et al.[7] show that networking applications typically have little ILP, and instead demonstrate a coarser packet-level parallelism. Packets are largely independent of each other and can be processed in parallel. This packet-level parallelism is not exploited by a single-threaded, single-CPU, super-scalar processor that looks for ILP in only one thread of execution. Many networking applications also do not demonstrate a high cache locality [6]; this is especially true for applications that handle a large number of flows, with packets belonging to the same flow spaced far away in time. Thus, a cache miss is not a rare event, and coupled with low ILP, can result in low processor utilization, as the processor spends idle cycles waiting for memory accesses to complete.

## 1.2 Network Processors for Link Emulation

### 1.2.1 An optimized architecture for packet processing

Network processors are programmable processors that use a multithreaded, multiprocessor architecture to exploit packet-level parallelism, have instruction sets geared towards efficient implementation of common networking tasks, and typically use hardware acceleration units to offload processor intensive tasks. The Intel IXP series of network processors [15, 17, 18] has between 6 to 16 independent RISC CPUs called microengines. Each microengine has 4 to 8 threads and supports hardware multithreading with low-overhead context swaps. Threads can start asynchronous memory and IO operations and can continue processing while waiting for the operation to complete. If there is no computation to be overlapped, the thread can yield control and let another thread process a different packet. This coarse-grained thread-level parallelism is more closely matched to the packet-level parallelism shown by networking applications, and can help hide high memory latencies. Figure 1.1 illustrates this parallelism by showing a snapshot of execution on one of the microengines of the IXP1200 network processor. As seen in the figure, in the ideal case, as long as there is at least one runnable thread, the latency of memory operations can be completely hidden behind the compute cycles.

The IXP series of network processors also provides hardware support for many of the common networking tasks like hashing, random number generation, buffer allocation and queuing. This enables efficient implementation of these tasks which helps to scale to large packet rates; especially since these tasks form the core of a

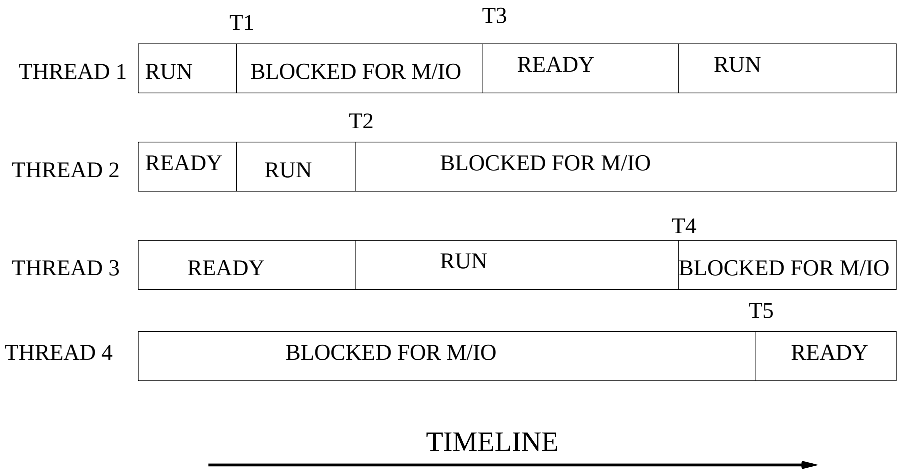


**Figure 1.1**. Thread-level parallelism on an IXP1200 microengine

wide variety of networking applications. In particular, a link emulator can use hashing for fast classification of packets into links, pseudo random number generators for implementing packet loss models and RED [10] queuing, and hardware support for buffer allocation and queuing for supporting large number of link queues and delay buffers. These processors also support low-latency hardware primitives for synchronization, which can help avoid expensive software synchronization, as packets and data structures are handled by multiple threads across multiple microengines in parallel.

## 1.3 Thesis Overview and Roadmap

This thesis proposes using network processors for building high capacity network link emulators. To evaluate our proposal, we have designed and implemented a link emulator, LinkEM, on the IXP1200 network processor. LinkEM's emulation model is based upon Dummynet's two-queue model [30]. However, it is a complete reimplementation in microengine assembly language on the parallel resources of the IXP1200 network processor. We discuss design techniques that we employed to map LinkEM tasks across the 6 microengines and 24 hardware threads of the network processor. We present a detailed evaluation of LinkEM, which includes validating its emulation accuracy, measuring its emulation throughput for a range of packet sizes, and measuring its link multiplexing capacity, and compare it with Dummynet on a comparable PC platform.[1]

The contributions of this thesis are:

- A description and demonstration of design techniques to map network applications to the IXP1200 network processor.
- A detailed description of the design and implementation of a link emulator on the IXP1200.

[1] The IXP1200 was released around 2000. We used an 850MHz PC with a 32-bit/33 MHz PCI bus which represented a reasonably high-end commodity PC platform around that time.

- Evaluation and comparison of the link emulator to a PC-based implementation in terms of emulation accuracy, throughput, and link multiplexing capacity.

The rest of this thesis is organized as follows. Chapter 2 lays out the background for the rest of the document. We describe Emulab, some important aspects of link emulation, and Dummynet's two-queue link emulation model. A network processor architecture, both hardware and software, is very different from a PC architecture, and it is crucial to understand the key differences between the two platforms. Therefore, in Chapter 3, we describe the architecture of the Intel IXP1200 network processor which we have used as the implementation platform in this thesis. Chapter 4 starts off by discussing the techniques used in designing LinkEM on the IXP1200. We then describe the different tasks in LinkEM, and their mapping across the multiple microengines and threads of the network processor. In Chapter 5, we present a evaluation of LinkEM and a comparison with Dummynet. Chapter 6 discusses related work, while Chapter 7 concludes this document.

# CHAPTER 2

# BACKGROUND

## 2.1 Emulab

Emulab is a time- and space-shared large-scale emulation environment used for networking and distributed systems research. Users specify a virtual topology graphically or via an ns script [33], and Emulab automatically maps it into physical resources that emulate the topology. PCs serve many roles in the testbed: as end-hosts in an emulated distributed system, as traffic generation and consumption nodes, as routers in the emulated topology, and as link emulator nodes. They are connected by Cisco switches which act as a programmable patch-panel, connecting the PCs in arbitrary ways to emulate the user-specified topology.

Emulab implements link emulation by inserting PCs running Dummynet into the topology. One of the important goals of Emulab is transparency; the end-user application or protocol should not need extensive modification to run on Emulab versus running directly on a real network.[1] Thus network emulation is provided with the minimum intrusion, and in case of link emulation, Emulab achieves it by configuring the link emulation nodes as Ethernet bridges, which transparently relay the packets between the end-hosts after emulation. Figure 2.1 shows a sample ns file specifying an experiment, and Figure 2.2 shows the associated virtual topology and its physical realisation in Emulab. In Figure 2.2, dn0 and dn1 are the link emulation nodes which emulate the desired link characteristics.

Emulab also supports changing network characteristics during experiment run-time through an event system based on the Elvin publisher-subscriber system [31]. For example, link characteristics like bandwidth and delay, as well as traffic

[1] In fact, this is one of the important advantages of emulation over simulation. A protocol once tested on a simulator typically involves substantial amount of porting to move it to the real world.

```
set ns [new simulator]  # Create the simulator
source tb_compat.tcl    # Add Emulab commands
$ns rtproto Static      # Static routing

set n0 [$ns node]        # create new nodes
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

set r0 [$ns node]
set r1 [$ns node]

$ns set duplex-link $n0 $r0 100Mb 10ms Droptail    #create links
$ns set duplex-link $n1 $r0 10Mb 25ms Droptail
$ns set duplex-link $n2 $r1 100Mb 10ms Droptail
$ns set duplex-link $n3 $r1 10Mb 10ms Droptail
$ns set duplex-link $r0 $r1 1.5Mb 100ms RED

$ns run                  #run on Emulab
```

**Figure 2.1**. Experimental script in Emulab

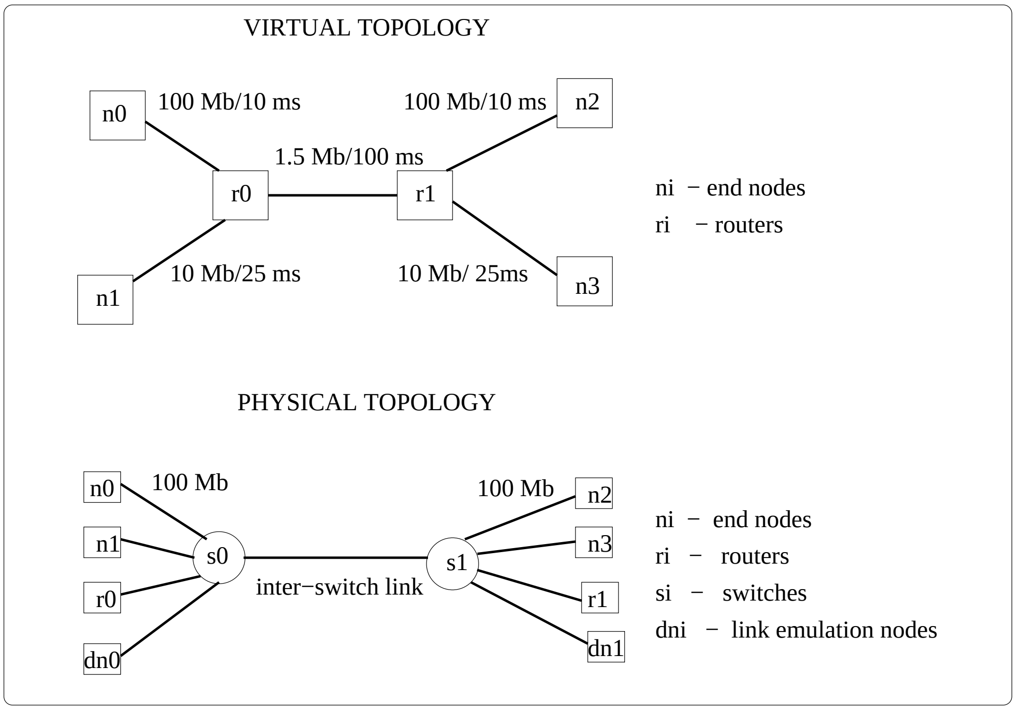


**Figure 2.2**. Topology mapping in Emulab

generator characteristics like packet size and packet rate can be changed during experiment runtime. These features bring much of a simulator's ease of use and control to emulation, while retaining its realism. In addition, through automated and quick experiment configuration, and support for automatic swap in and swap out, Emulab encourages an interactive style of experimentation, heretofore only attributed to simulation.

## 2.2 Link Emulation

Network links are of various types: wired or wireless, point-to-point or shared, and LAN or WAN links. Each type of link has its own unique characteristics, for instance, wireless links employ control packets for collision avoidance and have a channel interference model associated with them. However, some of the common link properties that can be abstracted across different types of links are *link bandwidth, link delay, link loss,* and *a link queuing model*.

Link bandwidth introduces queuing delays, when multiple flows compete for a link's capacity. For congestion reactive traffic like TCP, the end-node will reduce its sending rate to match its share of the the link bandwidth. However for "firehose" applications using UDP, packets are dropped in the link queue if the send rate continues to be higher than the link bandwidth. Link delays consist of three components: a queuing delay as packets wait to share a common link, a transmission delay that represents the time taken to put the entire packet on the wire, and a propagation delay or latency of the link. Link losses include queuing losses, as well as random bit errors on the links. A link queuing discipline could be FIFO or RED [10] for instance, and is typically used to absorb packet bursts, and for congestion control.

### 2.2.1 Aspects of link emulation

This section discusses some important aspects of link emulation that can be used as qualitative and quantitative metrics to describe a link emulator implementation.

- **Forwarding capacity:** This is the peak number of packets per second that the emulator can forward. It is largely independent of the packet size, because, other than the packet copying costs at the input and output of the emulator, all other processing involves only the packet header. Smaller packet sizes mean that more packets fit into the same bandwidth, increasing the amount of emulation processing required per link. Thus minimum-sized packets are the worst case load for link emulation, similar to routing.

- **Link emulation capacity:** This measures the maximum number of physical and multiplexed links that can be emulated with the desired accuracy. Since emulation runs in real time, the packet forwarding rate decides the number of emulated physical links. For example, an emulator that can forward a gigabit of traffic per second through emulation processing can essentially emulate a maximum aggregate emulation traffic of a gigabit.

  Many networking experiments use lower speed links than the available physical bandwidth; thus an ability to multiplex a number of such links on a physical link is useful. For instance, a gigabit of link bandwidth can be used to emulate six OC-3[2] links or a thousand 1 Mbps links or two thousand 512 Kbps DSL-like links.

- **Accuracy:** Like all timing related applications, the accuracy of an emulator depends on its timer granularity. For instance, an emulator using a 1 ms timer cannot accurately emulate links with bandwidth and delay that require sub-millisecond granularity. Coarser timer granularities result in the emulator adding burstiness to the emulated traffic; a 100 us timer for instance releases up to 15 minimum-sized Ethernet packets in one burst while emulating a 100 Mbps link.[3] In practice, this is generally not a problem, except for very fast (high bandwidth) and/or very short (low latency) links.

---

[2] 155 Mbps

[3] Inter-packet arrival time for minimum-sized Ethernet packets on a 100 Mbps link at line rate is 6.75 us. Thus burst size = 100 us / 6.75 us = 15.

Accuracy is also affected by the amount of traffic and the number of multiplexed links handled by the emulator. Under high loads, timer ticks can be missed, and since emulation tracks real time, this can lead to inaccurate emulation.

- **Extensibility:** This governs whether new models of link loss or new delay models or queuing disciplines can be added into the emulator without a significant redesigning effort. A component-based emulation environment for instance can offer extensibility, which can support adding new code modules into the emulator.

- **Feature Set:** This measures qualitatively the features that are provided by the emulator. Some emulators provide delay and loss models, others provide packet duplication, packet reordering and cross-traffic injection. Wireless link emulators provide models for channel interference and collision avoidance, while LAN emulators provide models for collision detection and exponential back off.

## 2.3 Dummynet

Dummynet [30] is a FreeBSD-based link emulator and traffic shaper. It is an in-kernel implementation that sits between the network drivers and the networking stack and can intercept packets both entering and leaving the node. Below, we discuss in brief, Dummynet's emulation model and its implementation inside FreeBSD.

### 2.3.1 Emulation model

A Dummynet abstraction for a link is called a pipe. A pipe has a configurable bandwidth, delay, and a loss rate associated with it. It is modeled by two queues: a bandwidth queue called an $R$ queue through which packets are drained at the link bandwidth rate, and a delay queue called a $P$ queue where packets are buffered for the link delay. The size of the $R$ queue and its queuing discipline (FIFO or RED) can be set by the user. The size of the $P$ queue affects the maximum delay-bandwidth product that can be supported for a link (see [30] for details).

### 2.3.2 Implementation

Dummynet relies on the FreeBSD IP firewall to filter packets and divert them to pipes. A user program called `ipfw` is used to create firewall rules that match packets to Dummynet pipes and to configure pipe characteristics. Dummynet can be hooked into the bridging or routing code in the networking stack of a FreeBSD kernel. In Emulab, Dummynet is configured to run in bridging mode, so that link emulation is transparent to end-nodes and applications. Figure 2.3 shows how the Bridging and Dummynet modules interact on Emulab link emulation nodes.

In the incoming path, packets are copied from the network interface to an mbuf chain, passed through the IP firewall, and diverted to Dummynet pipes. All this code runs in interrupt context, and network interrupts are turned off during this processing. A Dummynet callback function is triggered once every tick, which emulates the configured links. It is called at softint interrupt priority, however it turns off network interrupts during the entire processing to protect shared data structures from corruption. Once a packet is subjected to the link characteristics, it is sent to the bridging layer which looks up the destination interface, and the

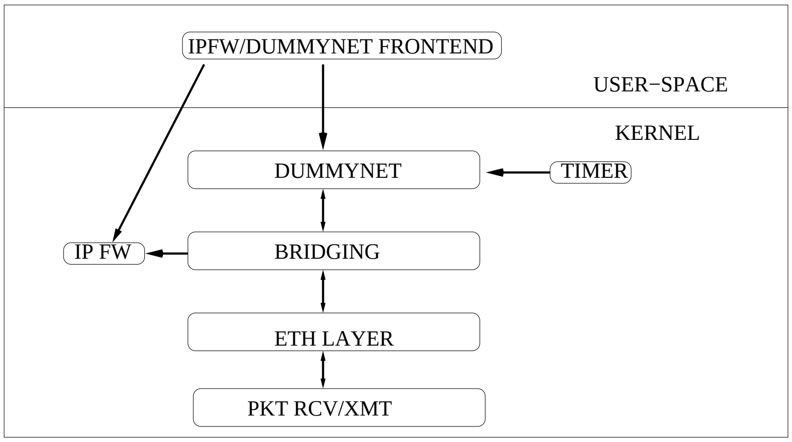


**Figure 2.3**. Dummynet in bridging mode

packet is then enqueued in the interface queue, ready to be sent out of the link emulator.

# CHAPTER 3

# IXP1200 ARCHITECTURE

In this chapter, we describe the hardware and software architecture of the IXP1200 network processor [15]. Unlike a PC platform, achieving high performance on a network processor requires intimate knowledge of the parallel processing engines and other hardware assists provided by the network processor. We describe the programmable processors, the on-chip resources like Scratchpad memory and the hardware hash engine, as well as the on-chip interfaces to off-chip resources like SRAM, SDRAM, and network interfaces. Since network processors tout flexibility through programmability, the software architecture on these processors is the other key part of this platform. We describe both, the software programming environment or development platform, and the component based programming model [16] of the IXP1200 network processor.

## 3.1 Hardware Architecture

Figure 3.1 shows the hardware architecture of the IXP1200 network processor, including the chip internals and the external interfaces.

### 3.1.1 Programmable processors

The IXP1200 consists of seven programmable engines, one StrongArm core and six microengines, all on the same die.

- **StrongArm Core:** The StrongArm is a 32-bit ARM general-purpose processor with configurable endianness that is clocked to run at 232 MHz. It has a built-in serial port, support for virtual memory, a 16 KB instruction cache, an 8 KB data cache and supports a byte-addressable interface to SDRAM and

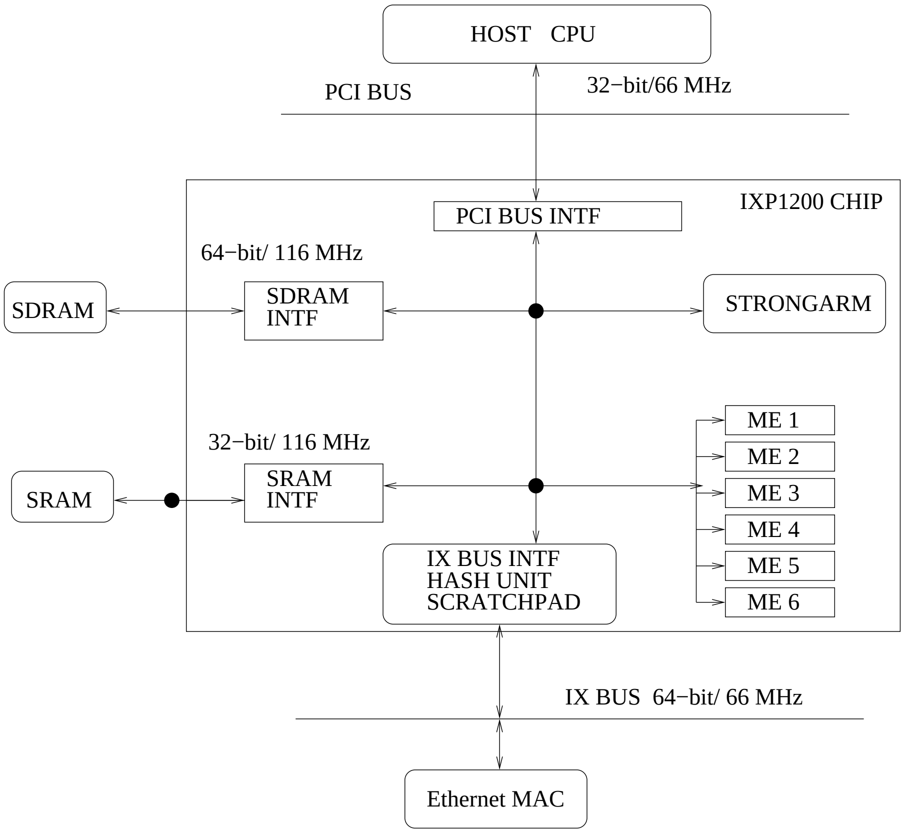


**Figure 3.1**. Architecture of the IXP1200 network processor

SRAM memories. Like other RISC processors, it supports a small set of simple instructions and a large register file. This processor is typically targeted to handle control plane processing and management plane applications.

- **Microengines:** The microengines are RISC processors optimized for data-plane packet processing; each microengine is clocked at 232 MHz. The key components of the microengine architecture are described below.

  - **Multiple hardware threads:** Each microengine supports four hardware threads. A thread owns a portion of the microengine's register set and has its own program counter. Thus context switches are very fast as they do not involve storing and loading state from memory. A non-preemptive scheduler switches between the threads; threads voluntarily swap out on memory and IO operations to let other threads run, thereby increasing microengine utilization. The microengine executes code from an on-chip 8 KB program store; each microengine has one such program store that is shared by all four threads.

- **Instruction pipeline:** A microengine has a five-stage pipeline divided into fetch, decode, fetch operands, execute, and write stages. Each stage takes one cycle to complete, thus the ideal clocks per instruction (CPI) is 1. Conditional branches can cause a stall in the pipeline, and the instruction set supports branch instructions that let the programmer control the path to fetch instructions from at a branch. Thread switches typically take about three cycles to flush the pipeline for the outgoing thread and to load the pipeline for the new thread.
- **Instruction set:** The instruction set supports operations that operate at bit, byte and long-word levels. The ALU supports arithmetic and logical operations with shift and rotate operations in single instructions. The microengines do not provide integer multiplication or division and do not support floating point operations. Integer multiplication can be accomplished through conditional iterative add operations based on sign condition codes.
- **Register set:** Each microengine has 128 32-bit general-purpose registers (GPRs), 64 SRAM transfer registers, and 64 SDRAM transfer registers. The GPRs can be accessed in a thread-local mode, in which each thread can access 32 registers, or in a global mode, in which all threads access the entire pool of registers. These modes are not exclusive, i.e., the programmer can use some registers as thread-local registers and designate others as global registers at the same time.

  Transfer registers are used to stage data while the data are being transferred between the microengines and memory. The advantage of having the GPR and the transfer register set different is that the GPRs can be used in computation, even as the transfer registers move data to or from memory, thus supporting asynchronous memory operations.

### 3.1.2 Functional interfaces

Functional interfaces on the IXP1200 consist of the SRAM and SDRAM interfaces, the FIFO bus interface (FBI) and the PCI interface.

- **SRAM interface:** The SRAM interface is an on-chip interface to up to 8 MB of off-chip SRAM. The SRAM memory has an unloaded latency of about 16-20 cycles and supports peak bandwidth of about 3.7 Gbps (32 bit/116 MHz bus). It occupies a middle ground between the on-chip Scratchpad memory and the off-chip SDRAM memory in terms of memory latency and size.

  SRAM is typically used for storing flow tables, routing tables, packet descriptor queues, and other data structures that need to be accessed frequently and with low latencies. It supports atomic bit-test-set and atomic bit-set-clear instructions that are useful for synchronization between microengines. It also supports CAM locks that can be used to lock regions of memory while updating shared data structures. The SRAM interface supports eight hardware push-pop queues that offer a fast atomic way to allocate and deallocate buffers across multiple microengines.

- **SDRAM interface:** The SDRAM interface supports up to 256 MB of off-chip SDRAM with a peak bandwidth of approximately 7.5 Gbps (64-bit/116 MHz bus) and an unloaded latency of 33-40 microengine cycles. SDRAM is typically used for storing packets; there is a direct data path from the SDRAM to the IX bus interface (network port interface) which supports fast data transfer between memory and the ports.

- **FIFO bus interface:** The FBI interface contains the following components: an IX bus interface, a hash unit which computes 48- and 64-bit hashes, a cycle count register which increments at the core frequency (232 MHz), and a 4 KB on-chip Scratchpad memory. The IX bus interface and microengine software together co-ordinate the transfer of packets between memory and the ports. The hash unit supports up to three independent hashes in a single instruction, and can be used to offload hash computation from the microengines. The cycle-count register provides a timer for implementing timing based tasks. The Scratchpad memory is the smallest latency memory with unloaded latency of about 12-14 cycles. It offers primitives like atomic increment which can be used to maintain statistics, and atomic bit-test-set and bit-test-clear instructions which can be used for fast inter microengine synchronization.

- **PCI interface:** The PCI interface allows the IXP1200 to connect to external devices like general-purpose CPUs.

## 3.2 Software Architecture

The software architecture on network processors is a key ingredient for the success of this technology. It is a challenge to present a development and programming environment that lets the developer peek into the hardware architecture to optimize for performance and at the same time provides an abstraction of the hardware to make software development easy. The software architecture of the IXP1200 provides a developer with familiar Linux-based compilation and debugging tools for the StrongArm, and Windows-based microengine compiler, assembler and debugger for the microengines. It also provides a programming framework, called the Active Computing Element (ACE) [16], for development of reusable software components that can be used across applications, and even across different generations of a network processor.

### 3.2.1 Development platform

The development platform on the IXP1200 consists of two main parts:

- **Linux-based cross-platform tool-chain for the StrongArm:** A Linux machine is used to host the cross-platform compilation and debugging tool-chain for the StrongArm processor on the IXP1200, which is the familiar `gcc` tool-chain with C and C++ cross compilers, linkers, and debuggers. The StrongArm boots up in embedded Linux, thus this platform is familiar to Unix developers.

- **Windows-based development environment for the microengines:** The Windows part of the development environment includes a microcode assembler, a C compiler, a debugger, and a cycle-accurate simulator for the IXP1200 microengines. The simulator is typically used by a software development team in the initial stages to develop and test the software parallel to the hardware board development. In the later stages, it can be used to optimize specific parts of the microcode for higher performance.

  Software for the microengines can be written in microcode, which is a structured assembly language, or can be written in a subset of ANSI C called MicroC. The C compiler for the microengines exports a familiar environment to programmers. It also exports some hardware features, which can be used by programmers to optimize their code. For example, the compiler supports primitives to control whether a variable should be stored in a microengine register, on-chip Scratchpad memory, or off-chip SRAM memory. The debugger is used to debug code running on the microengines on the IXP1200 hardware, or code running on the simulator.

### 3.2.2 Programming model

Networking applications typically operate in three planes: data, control and a management plane. The fast-path in the data plane handles most packets, e.g. normal IP packets. The slow-path handles exception packets like broadcasts or IP packets with options. The control plane handles control messages and creates, maintains, and updates the data and forwarding tables used by the data plane. Routing protocols like RIP and OSPF run in this plane. The management plane is responsible for configuration of the system, for gathering statistics, and for providing end-user interaction.

The software programming architecture on the IXP1200 provides a framework called Active Computing Element (ACE) [16] for creating applications in the data and control planes. Users can create reusable software components on both the StrongArm and the microengines, which can then be stitched together to create applications. At the highest level, the architecture is composed of two types of ACEs:

- **Core ACE or Conventional ACE:** These ACEs run on the StrongArm core of the IXP1200 processor. They are connected in a packet processing pipeline, with packets moving between the ACEs. These consist of two conceptual parts:
  - Classification: During this phase, the ACE applies a sequence of rules to the packet and places associated actions on a queue.
  - Action: During this phase, each action on the queue is executed. The actions typically involve processing the packet and then delivering it to the downstream ACE.

  Core ACEs are developed in C or C++ and a special language called Network Classification Language (NCL), which is used for writing the classification rules. ACEs also export an interface which can be used for configuring certain properties of the ACE, for example, an ingress ACE exports an interface to set the IP addresses of the interfaces and the interface MTU size. ACEs can be bound to each other to create a packet processing pipeline. The ACE runtime system provides this binding mechanism and also allows changing it dynamically to alter the packet flow.

- **MicroACE or Accelerated ACE:** MicroACEs consist of two types of components: slow-path or core components, which are regular ACEs that run on the StrongArm, and fast-path components, which run on the microengines. The microengine components handle the common packets in the fast-path, while exception packets are handed over to the core component for slow-path processing. A MicroACE has the following main architectural elements:

- Microblock: A microblock is a component running on the microengine. Examples of microblocks are an IP forwarding microblock or a bridging microblock. Microblocks can be written in microengine assembly language or in MicroC.
- Microblock Group: Multiple microblocks can be combined and downloaded as a single image on a microengine, these form a microblock group. For example, an ingress microblock, a bridging microblock and a queuing microblock can form a microblock group and can be downloaded on a microengine. Microblocks in a microblock group are statically bound at compile time.
- Dispatch Loop: A dispatch loop implements the packet flow between the microblocks in a microblock group. It caches commonly used variables in registers and provides macros for microblocks to access packet headers. Typically application functionality is encoded in the dispatch loop, thus insulating the microblocks from the application details, so that they can be reused across different applications.
- Resource Manager: This module runs on the StrongArm and provides an interface to the core component of an MicroACE to manage its microblock. Using the Resource Manager, the core component and microengines can share memory for data structures, and pass packets back and forth.
- Core Component: The core component of a MicroACE runs on the StrongArm, and appears as a conventional ACE to other ACEs in the system. It is responsible for allocating memory, setting up shared data structures, and patching the load time variables in its microblock. During runtime, it handles slow-path exception packets from its microblock.

MicroACEs can be combined along with Core ACEs to form a processing pipeline to implement networking applications. Figure 3.2 shows an example of an IP forwarding application using MicroACEs and conventional ACEs. The Ingress, IP forwarding and Enqueue microblocks are combined in a microblock group with a dispatch loop that implements the packet flow between them, and downloaded on one microengine. The Dequeue and Egress microblocks are combined into another microblock group and downloaded on a second microengine. The core components of the microblocks run as independent processes on the StrongArm and form a processing pipeline for slow-path processing. The Stack ACE is an example of a conventional ACE which acts as an interface between the microACEs and the Linux network stack running on the StrongArm.

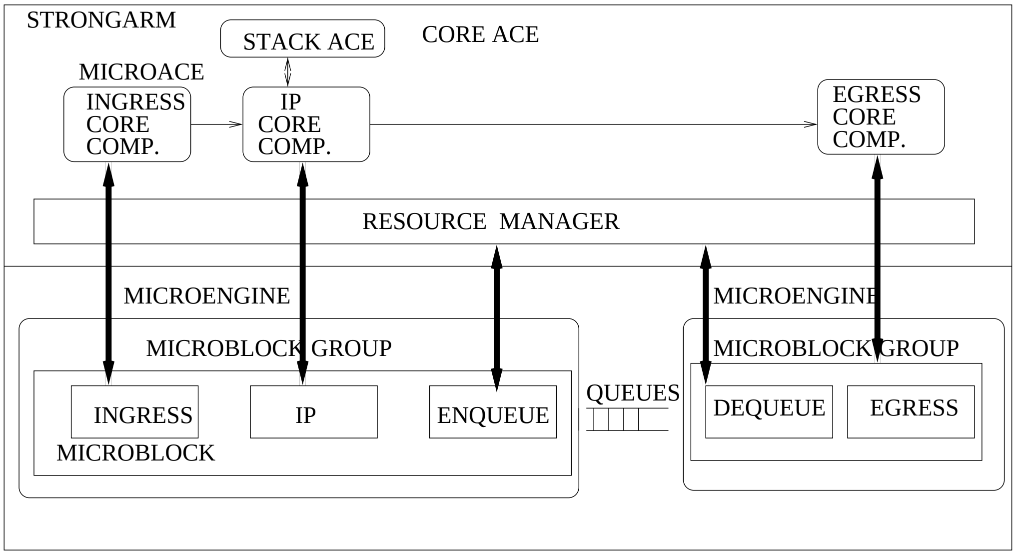


**Figure 3.2**. An example MicroACE IP forwarding application

# CHAPTER 4

# DESIGN AND IMPLEMENTATION

As described in Chapter 3, the IXP1200 network processor provides a large pool of parallel resources tuned to fast packet processing: 1400 MIPS across 6 parallel microengines, four hardware contexts on each microengine, instructions optimized for network tasks, a hardware hash unit, and hardware synchronization primitives. However, it is a significant challenge to map the application to these parallel resources on the IXP1200 to maximize performance, without running into communication and synchronization bottlenecks.

In this chapter, we first discuss some software design techniques that can be used to layer an application on the multiple threads and microengines of the IXP1200 network processor. We then describe the design of LinkEM: its emulation model, the tasks from which it is composed, and a mapping of these tasks across the six microengines and twenty-four hardware threads.

## 4.1 Software Design on the IXP1200

The MicroACE [16] architecture provides a way to build reusable software components on the StrongArm as well as on the microengines, and stitch them together to implement packet forwarding applications. However, the architecture does not provide any hints to the programmer about how to split an application into tasks and how to map tasks to use the available resources. This splitting and mapping is typically done manually by the programmer. In the following subsections, we discuss techniques for partitioning a group of tasks across microengines and threads that we employed while designing LinkEM on the IXP1200 network processor. These techniques are not the only way or even necessarily the best way of splitting tasks across microengines and threads. However, they worked well for us for designing LinkEM on the IXP1200.

### 4.1.1 Assigning threads on a single microengine to tasks

Each microengine on the IXP1200 processor has four hardware contexts that support coarse-grained multithreading under software control. One of the ways of using the multiple threads on a microengine is to perform the same task on all four threads; each thread acts on a different packet, and yields to other threads on memory and IO operations to maximize microengine utilization.

Assume that a task is executed by a single thread and takes $T_t$ cycles to complete: $T_t = T_c + T_m$, where $T_c$ is the compute cycles and $T_m$ is the memory and IO access cycles. This task can handle an inter packet arrival time $T_a$, if $T_a > T_t$ or $T_a > T_c + T_m$. As inter packet arrival times approach memory latencies, it is difficult to meet line rate because the processor spends large number of cycles waiting for memory operations to complete. To increase microengine utilization, we can execute the same task on all four threads, with each thread operating on a different packet. Let u be the fraction of memory and IO access cycles hidden by active thread computation when running multiple threads on a microengine. Figure 4.1 shows four packets being processed in parallel by four threads on a microengine.

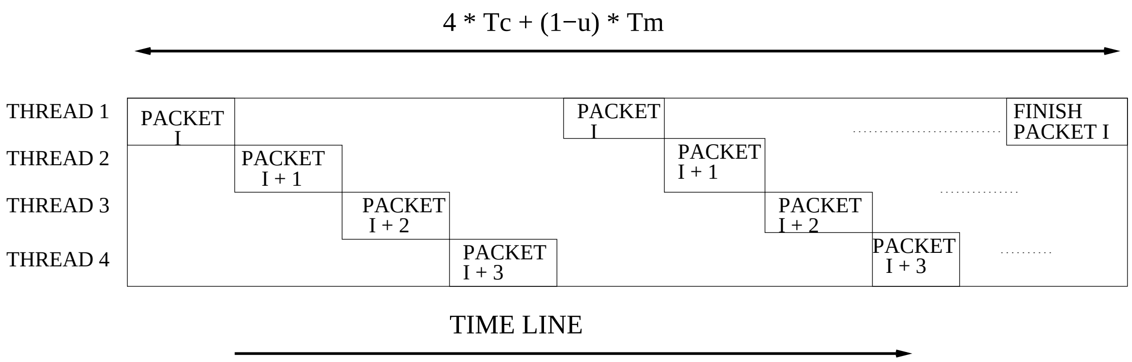


**Figure 4.1**. Exploiting multithreading on a microengine

Thus, each thread holds a packet for up to a wall-clock time of $T_t = 4 * T_c + (1 - u) * T_m$, out of which about $T_c$ cycles are spent processing a packet, while $(1-u) * T_m$ cycles are spent idle waiting for memory operations to complete. Since each thread now handles every fourth packet instead of every packet, the inter packet arrival time that can be handled by the combination of four threads is given by $T_a > T_c + (1 - u) * T_m$, which is lower than $T_c + T_m$, thus enabling a higher packet rate. In the ideal case, when the latencies of all memory and IO operations can be hidden by active thread execution, i.e., $u = 1$, the inter packet arrival time $T_a$ is bound by only the compute cycles, and is given by $T_a > T_c$.

### 4.1.2 Assigning microengines to tasks

Each microengine of an IXP1200 network processor offers about 232 MIPS; thus a single microengine often does not have enough compute power for a group of tasks which make up a packet processing application. In such cases, the group of tasks can be run on additional microengines to increase the available processing power.

- **Microengine pipelining:** A microprocessor pipeline uses multiple pipeline stages to process an instruction. Each stage handles a different instruction at a given time, and thus multiple instructions are overlapped in execution inside the pipeline. Similarly, a microengine pipeline can be used to partition a group of tasks, such that each task is executed on a separate microengine, handling multiple packets simultaneously in the pipeline.

  Assume that the compute cycles for a group of tasks is three times the inter packet arrival time. Let $T_c$ be the compute cycles to be applied to every packet. Then we can break down the group of tasks into a pipeline of three microengines, with each packet processed by three threads (one thread on each one of the three microengines). Since a microengine has four threads, there are a total of 12 packets simultaneously handled in the pipeline. Let task $t_i$ running on microengine $i$[1] have a fraction $u_i$ of its memory references

[1] slowest microengine pipeline stage

hidden behind active computation. Thus a thread on microengine $i$ holds a packet for a wall-clock time of $T_{t_i} = 4 * T_{c_i} + (1 - u_i) * T_{m_i}$ cycles, of which $T_{c_i}$ is the compute cycles spent on that packet and $(1 - u_i) * T_{m_i}$ is spent in waiting for memory accesses (see Section 4.1.1 above). Since a packet is handled by three threads, it gets a total microengine cycle time of $3 * T_{c_i}$ or $T_c$. This configuration meets a inter packet arrival time of $T_{c_i} + (1 - u_i) * T_{m_i}$. In the ideal case, if memory latencies are completely hidden and $u_i = 1$, then the inter packet arrival time supported has a lower bound of $T_{c_i}$. Figure 4.2 depicts a pipeline of three microengines implementing an Ethernet bridging application.

Apart from the compute cycles dictating the use of a microengine pipeline, code storage space and synchronization primitives also affect the splitting of tasks into microengine pipelines. For tasks that have a big code footprint, a separate microengine might have to be allocated, since the code space on each microengine is limited (about 2K instructions on the IXP1200). For example, in Figure 4.2, microengine 1 is dedicated to run the ingress task, microengine 2 runs the Ethernet bridging task and microengine 3 is dedicated for the egress task.

Running each task on a separate microengine also has the advantage that task specific state (or state independent of packets) can be cached in microengine

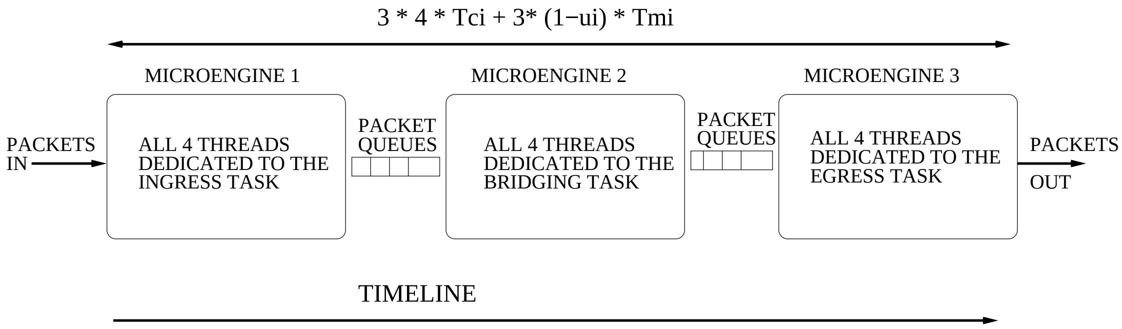


**Figure 4.2**. Using multiple microengines:pipelining

registers since it is only accessed by one microengine. For example, the Ethernet bridging task can cache frequently used entries of its bridging table in microengine registers to be accessed quickly. The flip side, however, is that packet headers or packet annotations like classification results have to be passed between microengines through expensive off-chip packet queues.

- **Microengine parallelism:** In this approach, a group of tasks run on a single microengine; this configuration is then executed in parallel on multiple microengines to get extra compute cycles. Figure 4.3 illustrates the Ethernet bridging application using microengine parallelism.

  Thus in our example, the collection of tasks with a compute cycle requirement of $T_c$ cycles is instantiated on three microengines, with each microengine running four threads. A packet is handled completely inside one microengine by one thread. Therefore, a single thread spends a wall-clock time of $T_{t_i} =$

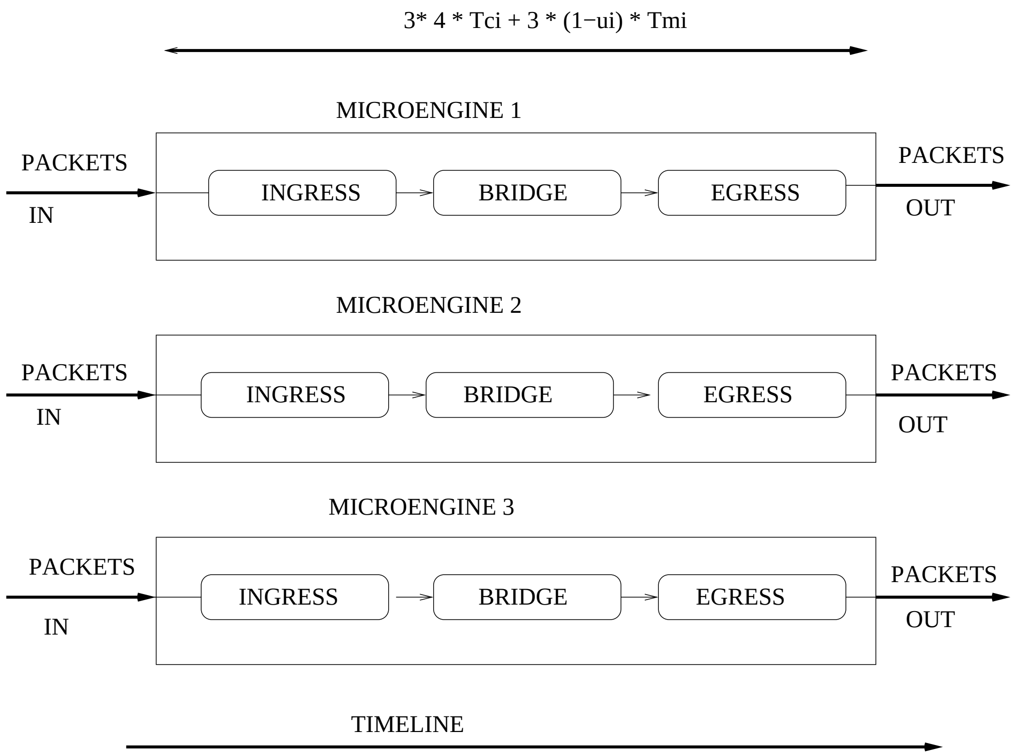


**Figure 4.3**. Using multiple microengines:parallelism

$3*4*T_{c_i}+3*(1-u_i)*T_{m_i}$ cycles, out of which $3*T_{c_i}$ or $T_c$ are the compute cycles spent per packet, and $3*(1-u_i)*T_{m_i}$ is the time spent waiting for memory operations. However, since each thread handles every 12th packet, this configuration can meet inter packet arrival time of $T_a > T_{c_i}+(1-u_i)*T_{m_i}$. In the ideal case, if memory latencies are completely hidden and $u_i = 1$, then the inter packet arrival time supported has a lower bound of $T_{c_i}$.

Each microengine now runs the entire group of tasks, and this configuration is replicated on three microengines. Every packet is handled completely by one thread on any one microengine. This means that packet headers once loaded from memory can be cached in registers, and passed on to the different tasks very cheaply. Similarly, packet annotations like classification results can also be cached in registers. However, since all tasks run on more than one microengine, any updates to shared state across microengines has to go through inter-microengine synchronization which can be prohibitive.

In Section 4.2.2, we describe how these techniques are used to map LinkEM to the microengines and threads of the IXP1200 network processor.

## 4.2 LinkEM Design

LinkEM's emulation model is based on the Dummynet two-queue model (described in Chapter 2); however it is a complete reimplementation in microengine assembly language on the microengines of the IXP1200 network processor. It supports a per-link configurable bandwidth, latency, loss and a queuing model. It also supports multiplexing lower speed links onto physical links; for example, multiple 512 Kbps links can be multiplexed on a 100 Mbps physical link. This feature is useful for modeling networking experiments which use a large number of low or moderate speed links to emulate WAN links. Incoming packets are classified early on into different links using source and destination IP addresses as the key. Once attached to a link, a packet is subjected to queuing, transmission and propagation delays configured for that link. It might also be dropped or forwarded, based on the link loss rate, or can be dropped if the link queue is full.

The emulation task in LinkEM is timer-triggered. Emulation time is maintained as a counter that is incremented once every tick.[2] The emulation time and the link characteristics are used to calculate the number of ticks packets have to wait, to model the different kinds of link delays. Since LinkEM can be used to potentially handle hundreds of links every tick, an efficient way is needed to handle the scheduling of all links. The obvious way of maintaining a linear list of all emulated links and traversing it every tick does not scale well as the number of emulated links increase.

Like Dummynet, LinkEM instead uses a priority queue for maintaining scheduling information of all active link queues in the system. The key for the priority queue is the expiration time for the first packet of a link queue. LinkEM uses a heap data structure to implement the priority queue. There are two types of heaps: a heap which contains all nonempty link bandwidth queues, and a heap containing all nonempty link delay queues. At every tick, LinkEM processes all link bandwidth and link delay queues whose expiration time is in the past as compared to the current emulation time. Packets from the bandwidth queue are moved to the delay queue for link latency emulation once subjected to queuing and transmission delays, while packets from the delay queue are moved to the output port queue for transmission out of the link emulator.

### 4.2.1 LinkEM tasks

At the highest level, LinkEM is split into 10 tasks: Ingress, Bridging, Classification, Queuing, Link Queue Scheduling, Timing, Link Loss emulation, Link Bandwidth emulation, Link Delay emulation, and Egress. Each of these tasks is implemented as a MicroACE. The StrongArm components handle initialization of data structures shared with the microblocks. In the case of Bridging, Classification and Egress, they also handle packets in the slow-path. The microblocks handle the fast-path on the microengines. We reuse the Ingress, Bridging and Egress microACEs from the Intel reference design code [16], while all other microACEs

[2] LinkEM is configured to run at 8192Hz, thus one tick is about 120 us.

are implemented from scratch. This section describes each of the microblocks and the data structures that it uses.

- **Ingress:** Figure 4.4 presents pseudo-code for the ingress microblock.

  The ingress microblock is responsible for receiving packets from the network ports and assembling them into memory. To implement this functionality, the microblock uses an on-chip hardware unit called the IX bus interface

```
start#:
load input port state
enter critical section
check for port readiness
leave critical section

if (data){
        enter critical section
        issue a request to transfer data from port to rcv buffer
        wait for transfer to complete
        leave critical section
} else br[start#:]

if(start of packet){
        allocate packet buffer
        Move 64-byte chunk from rcv buffer to microengine registers
}

Move 64-byte chunk from rcv buffer to SDRAM packet buffer
Update packet reassembly state

if(end of packet){
        Pass buffer handle to processing task
} else
        Pass NULL buffer handle to processing task

br [start#]
```

**Figure 4.4**. Ingress microblock

(see Figure 3.1), which connects to the external MAC ports over the IX bus. The IX bus interface contains sixteen 64-byte receive buffers, a small programmable engine called a ready-bus sequencer, and control and status registers (CSRs). The ready-bus sequencer polls the ports for incoming packets at the programmed interval, and updates the status registers about data availability. The ingress microblock in turn polls the status registers, and instructs the IX bus interface to transfer data from the port to the receive buffers. The IX bus splits the packet in 64-byte chunks, and each chunk is transferred separately to receive buffers and then to memory.

The ingress microblock is also responsible for allocating a buffer descriptor and the corresponding buffer to hold the packet. It assembles the 64-byte chunks into an entire packet in the buffer, and updates the buffer descriptor with meta information like the incoming port, the timestamp of the received packet, and the ingress queue number corresponding to the incoming port. The IXP1200 SRAM unit supports a hardware stack of buffer descriptors in SRAM which can be used to atomically allocate and deallocate buffers. Each SRAM buffer descriptor corresponds to a packet buffer in SDRAM where the actual packet is stored.

- **Bridging:** The bridging microblock is used to determine the output interface to which the packet should be forwarded after it is subjected to link emulation. The bridging table is implemented as an open hash table in SRAM, with the destination MAC address in the packet header as the hash key. The microblock loads the packet header into registers, extracts the 48-bit MAC address, and uses the hardware hash unit to produce a 48-bit hash. This hash value is then folded to index the hash table to access the bucket. The bucket contains a pointer to a linked list that is traversed to search for a matching MAC address. The search returns with the output interface to which the packet should be forwarded, which is updated in the packet meta data. The packet is then forwarded to the next processing block.

Packets with source and destination addresses that match in the bridging table are processed in the fast-path. If the destination address is not found in the table, then the packet is forwarded to the core component for flooding to all interfaces. A packet whose source address is not found in the table is also sent to the core so that its address can be added to the table. Similarly, broadcast, multicast, and ARP packets are diverted to the core for slow-path processing. Figure 4.5 shows pseudo-code for the bridging microblock.

- **Classification:** The classifier microblock is used to classify or demultiplex packets on the same physical link into constituent emulated links, so that packets can be subjected to the configured link characteristics. It implements exact match classification on a key consisting of source and destination IP addresses in the packet. Classification returns the ID of the link to which the packet belongs or returns failure if the packet is not mapped to any link. The

```
start#:

Use buffer handle to locate the packet buffer in SDRAM
Load packet header and extract the src and dst MAC addresses

Do RFC bridge checks to see if the src/dst addresses are valid

Hash the src/dst MAC addresses
Search the Bridging table for src/dst MAC entries

If (entry not found or learning needed){
        generate exception to bridging core component
        br [start#:]
}

Update the output port in the packet meta data

br[start#:]
```

**Figure 4.5**. Bridging microblock

link ID and packet descriptor are then forwarded to the next microblock if there is a match in the classification table.

The classification table is implemented as an open hash table in SRAM; the hardware hash unit is used to generate a 64-bit hash value from the 64-bit source and destination address key, which is subsequently folded to the table index size. The bucket at the index contains a pointer to a linked list that is searched for a match. Packets that do not match any link are forwarded to the core component for slow path processing. Figure 4.6 shows the pseudo-code for the classifier microblock.

- **Queuing:** The queuing microblock implements a separate queuing discipline per link. Queues are implemented as circular buffers in SDRAM, while queue descriptors are maintained in SRAM. This microblock uses the link ID passed in by the classifier microblock to locate the queue for the link. If the queuing discipline is FIFO, the packet is enqueued if there is space in the queue, otherwise it is dropped. If the link queue supports RED, then the

```
start#:

Use buffer handle to locate the packet buffer in SDRAM
Load packet header and extract src/dst IP addresses

Hash the 64-bit key
Fold it into an index of the required size
Search in the classifier table for a matching link

If(not found){
        generate exception to the core component
        br [start#:]
}
Pass on the link ID to the next component
br [start#:]
```

**Figure 4.6**. Classifier microblock

RED algorithm is used to determine if the packet is enqueued or dropped [10]. Currently, LinkEM does not implement the RED queuing algorithm. Figure 4.7 shows pseudo-code for this microblock.

- **Link Queue Scheduling:** As described in Chapter 2, a packet experiences two kinds of delay in a link bandwidth queue: a queuing delay sitting in the queue behind other packets and a transmission delay that models the time it takes for the packet to be put on the wire. If the queue is empty before the packet is inserted, then the packet does not experience any queuing delays. In this case, the link scheduling microblock computes the number of ticks that the packet should spend in the queue depending on the link bandwidth. With this as the key, the microblock then schedules the link in the bandwidth heap to be processed in the future by the bandwidth emulation task. It then stores the current emulation time in the queue meta information.

  If the bandwidth queue was nonempty before this packet was enqueued, then the queue is not scheduled. Note that in this case, the queue will already be in the bandwidth heap with its key as the expiration time of the first packet in the queue. The link bandwidth heap is shared between this microblock and the bandwidth emulation microblock. While this microblock is packet stimulated, the bandwidth emulation microblock runs at the configured timer granularity. Since both microblocks update the heap, a critical section is used

```
start#:

Use link ID to load the link descriptor and access the queue number
Enqueue the packet in the link bw queue by running FIFO/RED algorithm

br [start#:]
```

**Figure 4.7**. Queuing microblock

to protect insertions and deletions in the heap. Figure 4.8 shows pseudo-code for this microblock.

- **Bandwidth Emulation:** Figure 4.9 shows the pseudo-code for the bandwidth emulation task. At every timer tick, this microblock processes the bandwidth heap. It does a DelMin operation on the heap to get the link bandwidth queue with the minimum key. If the queue's expiration time is in the past as compared to the current emulation time, the queue is processed. Using the link descriptor handle from the queue's meta info, this microblock loads the descriptor and gets access to the link parameters.

```
start#:

If (link bw queue was not empty before this enqueue){
        /* The packet will also experience a queuing delay*/
        /* The queue is already present inside the heap */
        jump[start#:]
}

/* link queue was empty before this packet was enqueued*/
/* schedule this queue based on the transmission delay
for this packet*/

Use packet size and link bandwidth to determine number of
ticks to wait.

Insert the queue in the bw heap with the computed key

Store the current emulation time in the queue meta info,
so that the emulation task will know when this queue was
scheduled

br [start#]
```

**Figure 4.8**. Link queue scheduling microblock

```
wait_timer_tick#:
wait_inter_thread_signal()

process_bw_heap#:
Get MinKey from the bandwidth heap
Get current emulation time

if (Minkey > current_em_time || heap is empty){
        /* no queues to process this tick */
        br [wait_timer_tick#]
}

DelMin and get Min queue handle and key
Load queue meta info and get the link descriptor handle
Load link descriptor and get the bw and other link params

Update accumulated credit based on current time and
the time when the queue was scheduled

While (packets in the bw queue){
        QPeek(nextpktsize)
        If (packet size < credit){
                move packet to delay queue
                decrement credit by packet size
        } else
                break;
}
if (queueisempty){
        /* done processing this queue, see if any other
        queue needs processing */
        br [process_bw_heap#]
}

/* This queue needs to be scheduled again */
Compute new key for the queue
Insert queue in bw heap
br [process_bw_heap#]
```

**Figure 4.9**. Bandwidth emulation microblock

Based on the current emulation time, the emulation time when the link was scheduled, and the link bandwidth, it calculates the accumulated credit. Packets are then moved from the link bandwidth queue to the link delay queue based on the accumulated credit. For each packet, its expiration time from the delay queue, based on link latency, is calculated, and updated in the packet meta-data.

If the link queue still has packets that cannot be processed in this tick, the new expiration time for the queue is calculated based on the size of the first packet in the queue, the remaining credit, and the link bandwidth. The queue is then scheduled back into the bandwidth heap to be processed at a future timer tick.

- **Delay Emulation:** At every timer tick, this microblock processes the delay heap. It does a DelMin operation on the heap to get the link delay queue with the minimum key. If this key is in the past compared to the current emulation time, the queue is processed. Packets from the delay queue with expiration time in the past are moved to the output port queue. If the queue is not yet empty, it is scheduled back in the delay heap for processing at a future tick, with the output time of the first packet in the queue as the key. Figure 4.10 shows the pseudo-code for this task.

- **Loss Rate Emulation:** This microblock implements uniform packet loss rate for a link. Using the link descriptor handle, the microblock loads the link loss rate.[3] For every packet a random value is generated; if the value is less than the loss rate value, the packet is dropped, else the packet is forwarded.

  The IXP1200 does not provide a pseudo random number generator, nor does it provide instructions to make this job easy on the microengines. So to implement loss rate, the core component of this task generates the random

[3]The user configured loss rate value between 0 and 1 is scaled to an unsigned integer and stored in the link descriptor.

```
wait_timer_tick#:
wait_inter_thread_signal()

process_delay_heap#:

Get MinKey from the delay heap
Get current emulation time

if (Minkey > current_em_time || heap is empty){
        /* no queues to process this tick */
        br [wait_timer_tick#]
}

DelMin and get Min queue handle and key

While (packets in the delay queue){
        QPeek(nextpktexpirationtime)

        if(expiration time in the past){
                Find the output port queue for this packet
                enqueue the packet in the output port queue
        } else
                break;
}
if (queueisempty){
        /* done processing this queue, see if any other
        queue needs processing */
        br [process_delay_heap#]
}

/* This queue needs to be scheduled again */
Compute new key for the queue
Insert queue in delay heap

br [process_delay_heap#]
```

**Figure 4.10**. Delay emulation microblock

values and fills up a table in memory. This table is shared with the microblock; the microblock simply does a lookup in this table to get the random value and uses it to implement link loss. This technique works reasonably well, although not as well as generating random numbers since the table size is limited and random values start repeating depending on the table size. Figure 4.11 presents pseudo-code for this task.

The alternative, which is generating random values on the microengine, is very expensive in microengine cycles to do in the fast-path. The second generation of IXP processors includes an on-chip pseudo random number generator that can be used for this task.

```
start#:

Load the link loss rate
if (loss rate == 0){
        /* pass the packet */
        pass the buffer handle to next processing task
        br [start#:]
}

/* non-zero loss rate */
load the next random value from the random number table

if (random_value < loss_rate){
        /* drop the packet */
        release packet buffer
        pass NULL handle to next task
        br [start#:]
}

/* pass the packet */
pass the buffer handle to next processing task
br [start#]
```

**Figure 4.11**. Loss rate emulation microblock

- **Timing:** The IXP1200 has a 64-bit cycle-count register located in the FBI interface (see Section 3.1), which increments at the core frequency of 232 MHz. The timing microblock can be implemented to read this register in a loop, and generate signals to the emulation threads at the configured tick rate. However, the cycle-counter is co-located with the Scratchpad Memory and the IX bus interface that controls the movement of packets between memory and ports, so frequent reading of the cycle-counter adversely affects the access latencies of these hardware units. As a result, we decided not to use the cycle counter for implementing the emulation timer.

  Figure 4.12 shows the pseudo-code code for the timer microblock. As described in Section 4.2.2, all other microblocks use up only four of the six microengines. So we decided to dedicate one microengine to the timer microblock, and generate accurate timing without causing any load on the rest of the system. We run the timing microblock in one thread on the fifth microengine; this thread runs a simple loop with a preconfigured loop count based upon the tick value. Since this thread uses the microengine completely, we can calculate the loop count based on the microengine frequency to implement

```
start_timer_loop#:

load num_cycles to loop
Account for a constant number of cycles to implement the loop
while (num_cycles > 0){
        decrement num_cycles
}
;signal the emulation threads
inter_thread_signal(emulation threads);

br[start_timer_loop#]
```

**Figure 4.12**. Timer microblock

the configured timer tick value. When the thread exits the loop, it sends an inter-thread signal to the emulation threads and goes back into the loop.

- **Egress:** Figure 4.13 show pseudo-code for the egress task. Once packets are subjected to their link characteristics, the emulation microblocks enqueue the packets in output port queues. The egress microblock is then responsible for moving packets from the output port queues to the network ports for transmission. Just like the ingress task, the egress task uses the on-chip IX bus interface unit for this purpose. The IX bus interface contains a small programmable unit called the ready-bus sequencer, sixteen 64-byte transmit buffers, and control and status registers (CSRs). The ready-bus sequencer and the egress microcode cooperate to use the transmit buffers as a circular queue for staging packets temporarily, as they are moved from memory to output port buffers.

```
start#:
Get a packet from the output port queue

handle_next_chunk#:
Compute the xmt buffer to which this 64-byte chunk should
be transferred

Wait for xmt buffer to become free
move 64-byte chunk from memory to xmt buffer

check if port is ready for data
Validate xmt buffer so that chunk will be transferred to port

If (end of packet){
        release packet buffer
        br [start#]
}
br [handle_next_chunk#]
```

**Figure 4.13**. Egress microblock

The ready-bus sequencer polls the ports at regular intervals and updates the status registers whenever there is space in the port buffers. These status registers are in turn polled by the egress microblock. The egress microblock transfers the packet in 64-byte chunks to the transmit buffers. It then writes the output port and the offset of the chunk in the whole packet to the control words associated with the buffer. Once it validates the buffer, the transmit state machine moves it over the IX bus to the port for transmission.

### 4.2.2 Mapping LinkEM tasks to the IXP1200 microengines

In this section, we describe how LinkEM tasks (described in Section 4.2.1) are combined into microblock groups and instantiated on the microengines. Our aim is to produce *a* mapping of LinkEM tasks on the IXP1200 microengines, not necessarily the most optimized mapping, that helps us to evaluate the network processor for high-capacity link emulation. Figure 4.14 shows the high-level mapping of LinkEM on the six microengines of the IXP1200.

Microengines 1, 2, 3, and 4 form a microengine pipeline in which packets are processed as they pass through the link emulator. For tasks on microengines 1 and

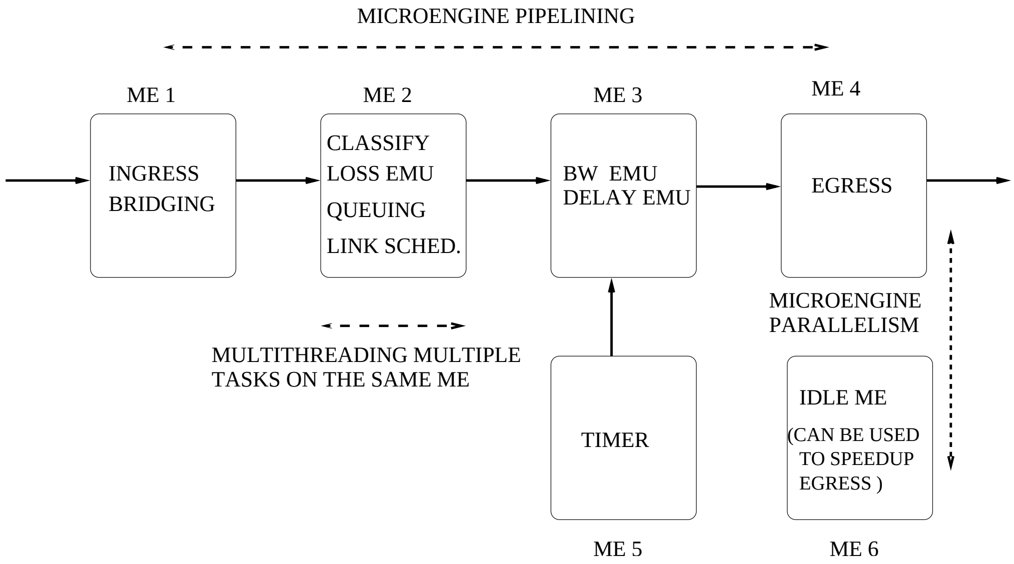


**Figure 4.14**. High-level design of LinkEM on the IXP1200

2, we estimate $T_c$ and $T_m$, and use the design technique described in Section 4.1.1 to group them. $T_c$ is estimated through a pseudo-code analysis of the code, while $T_m$ is obtained by multiplying the number of accesses by a task per packet by the unloaded memory and IO access latencies. Tables 4.1 and 4.2 show the latencies we measured for common transfer sizes and hash references typically done by the tasks. For tasks on microengines 3, 4, and 5, we employ qualitative analysis like the code footprint size, the task functionality, the amount of state shared with other tasks, and whether tasks are packet stimulated or timer triggered, to combine them into groups. The sixth microengine is idle in the current design. In Appendix A, we describe a design in which the sixth microengine can be used to speed up egress. The two-microengine design of egress is an example of microengine parallelism design technique.

Note that actual memory and IO latencies vary during runtime, as they depend on the load on the system, but we use unloaded latencies as a ballpark to create an initial mapping of tasks. Subsequent implementation and cycle measurements could be used to distribute tasks in a different way based on the measured bottlenecks in the system. However, producing an optimal mapping of tasks to microengines is not the focus of this thesis.

**Table 4.1**. Measured SRAM/SDRAM latencies for common transfer sizes (cycles)

| Memory Op. | 4 bytes | 8 bytes | 12 bytes | 16 bytes | 24 bytes | 32 bytes |
|---|---|---|---|---|---|---|
| SramRD | 18 | 22 | 22 | 26 | - | - |
| SramWR | 18 | 22 | 22 | 26 | - | - |
| SdramRD | - | 50 | - | 50 | 54 | 54 |
| SdramWR | - | 38 | - | 38 | 42 | 42 |

**Table 4.2**. Hash unit latencies for computing 48- and 64-bit hashes

| Hash Operation | Num of cycles |
|---|---|
| Two 48-bit hashes | 40 |
| One 64-bit hash | 35 |

- **Microblock Group 1:** The ingress and bridging microblocks are combined into a group and instantiated on one microengine of the IXP1200 processor. Table 4.3 shows the compute and memory cycle requirements for both these microblocks.

  If we execute this group of tasks on four threads of one microengine with each thread handling a different packet, then a thread can hold a packet for up to four packet arrival times. The IXP1200 board that we use as our implementation platform has four 100-Mbps interfaces, thus the inter packet arrival time for each port is about 6.75 usecs for minimum-sized Ethernet packets. Since each thread handles one port (this design avoids sharing and synchronization of packet reassembly state between threads), it can hold the current packet for about 6.75 usecs or 1550 cycles (at 232 MHz microengine frequency) before the next packet arrives at its assigned port.

  The wall-clock time needed by a thread to process one packet is about $4 * T_c + (1 - u) * T_m$, or between 1200 and $1200 + (1 - u) * 500$ cycles. This can just fit within the 1550 cycle budget or overshoot it, depending on how many memory and IO cycles can be hidden behind compute cycles. Thus, a single microengine with four threads seems to be a good match for running the ingress and bridging microblock group.

  Figure 4.15 shows the ingress and bridging microblock group with the main data structures. Since the link emulation processing tasks run on other microengines, packet buffer handles and packet state are passed to them through memory. This is done by maintaining one packet queue per input

**Table 4.3**. Cycle counts for Microblock Group 1

| Task | Compute cycles | Memory-hash cycles | Total cycles per packet |
|---|---|---|---|
| Ingress | 80 | 195 | 275 |
| Bridging | 210 | 275 | 485 |
| Total | 290 | 470 | 760 |

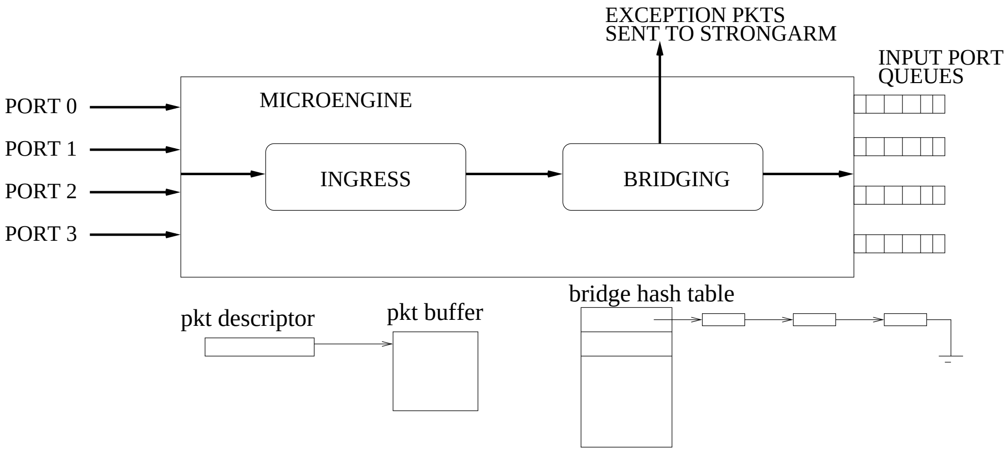


**Figure 4.15**. Microblock Group 1

port for holding the packet buffer handle and other packet annotations. Once the packet is assembled by ingress in memory and its output port is looked up by the bridging task, the packet handle and state is enqueued in this packet queue.

- **Microblock Group 2:** The classification, link loss emulation, queuing, and link queue scheduling microblocks are combined into a single microblock group and instantiated on one microengine. Table 4.4 shows estimated compute and memory/hash cycles based on a pseudo-code analysis of these tasks.

  This microblock group receives packets from the ingress and bridging microblock group (Microblock Group 1) through packet queues in memory. Small differences in execution speed between the two microblock groups can

**Table 4.4**. Cycle counts for Microblock Group 2

| Task | Compute cycles | Memory-hash cycles | Total cycles |
|---|---|---|---|
| Classifier | 40-50 | 200 | 240-250 |
| Loss rate em | 30-35 | 40 | 70-75 |
| FIFO Enqueue | 20-30 | 135 | 155-165 |
| Link schedule | 45-50 | 140 | 185-190 |
| Total | 135-165 | 515 | 650-680 |

be absorbed by the packet queue, but in principle, this microblock group has to run as fast as the ingress and bridging microblock group. Thus with four threads executing this group of tasks on a microengine, each thread can hold on a packet for up to one inter packet arrival time (on one port), or about 1550 cycles, before it has to handle the next packet.

From the table, we can see that the minimum wall-clock time needed to process each packet by a thread is about $4 * T_c + (1 - u) * T_m$, or between 660 and $660 + (1 - u) * 515$ cycles, which fits within the 1550 cycle budget. Note that the actual memory latencies at runtime will vary depending on the load on the system. Also, the running time of operations on the different data structures might vary depending on the packet being currently processed. For instance, certain buckets of the hash table might contain longer list of elements. Figure 4.16 shows this microblock group and the associated data structures.

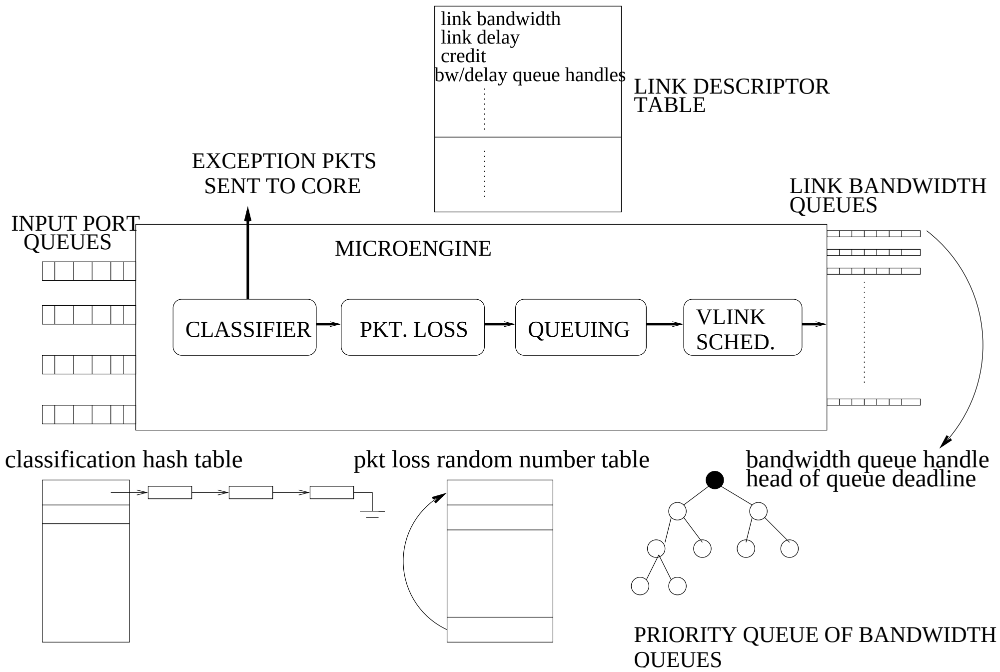


**Figure 4.16**. Microblock Group 2

Since these tasks run on the same microengine, packet headers and packet annotations like classification results are passed between tasks cheaply through microengine registers. Once loaded by the loss rate task, the link descriptor is passed through microengine registers to the link scheduling task, thus avoiding extra external memory lookups.

- **Microblock Group 3:** While the microblocks discussed until now were all packet-stimulated, the link bandwidth and link delay emulation microblocks are timer-triggered. They are scheduled once every tick and process all links as explained earlier (Figures 4.9 and 4.10). These microblocks have two kinds of costs: a per packet cost and a per link processing cost; every packet enters and exits the bandwidth, delay and transmit queues, and every link is scheduled both in the bandwidth and the delay heaps. Thus these are highly compute and memory-intensive microblocks. Figure 4.17 shows this group and its associated data structures.

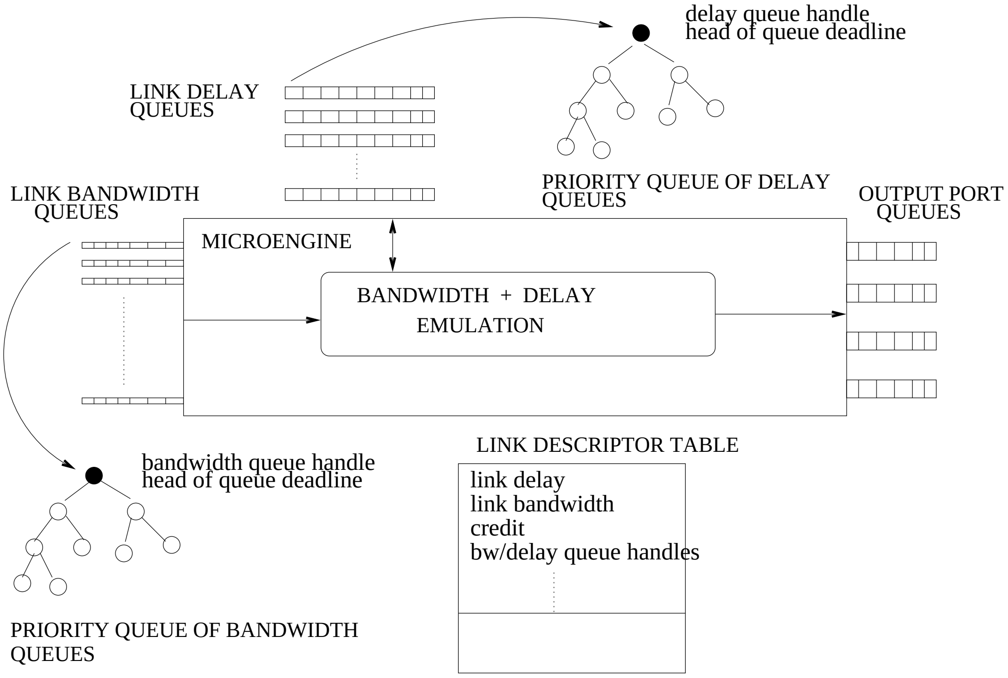


**Figure 4.17**. Microblock Group 3

The rationale for combining these two microblocks into one group is slightly different from before; they are both timer-triggered, functionally similar, and share a lot of state to which access can be synchronized cheaply using intra-microengine synchronization primitives. The link scheduling microblock from Microblock Group 2 schedules link queues in the link bandwidth heap. Since this microblock group also needs to access these data structures, and the two run on different microengines, we have to use inter microengine synchronization primitives to protect access to the link queues and the link bandwidth heap.

- **Microblock Group 4:** Packets once subjected to emulation, are enqueued in output port queues. The egress task is responsible for dequeuing packets from the output port queues, and moving them to the ports for transmission. It is functionally independent from the rest of the emulation tasks, and is best implemented as a separate microblock group on its own microengine. Since this microblock is highly hardware dependent,[4] its code changes significantly between different versions of the network processor. Implementing it on a separate microengine insulates other microblocks that are hardware-independent from changes in the hardware.

- **Microblock Group 5:** As described in Section 4.2, the timing microblock uses a simple microengine instruction loop and a loop count based on the required timer value to implement timing. A single microengine thread executes this loop, without ever giving control to any other thread on that microengine. Hence, the timing microblock forms its own group, and has a dedicated microengine in this design. This is an example where the functionality or implementation details of the microblock force it into a group of its own on a separate microengine.

- **Microblock Group 6:** The sixth microengine is idle in the current design.

[4] On the second generation IXP2400 processor, the hardware interface for transmission over the ports is different from the interface on the IXP1200

# CHAPTER 5

# EVALUATION

In this chapter, we give a detailed evaluation of LinkEM that includes validating its accuracy, measuring its packet throughput capacity for different packet sizes, and measuring its link multiplexing capacity for a range of low and medium bandwidth multiplexed links.

## 5.1 Accuracy Validation

### 5.1.1 Bandwidth emulation

To measure LinkEM's bandwidth emulation accuracy, we configured it to emulate different bandwidths and compared them with measured bandwidths using end-node traffic generation and measurement tools. We used the `iperf` [26] traffic generation tool to generate TCP traffic. `Iperf` was also used as a measurement tool; `iperf` receivers were configured to measure the bandwidth that the connections experience over the desired time interval.

We connected four end-nodes, one to each of the four ports of the emulator, and configured two end-nodes to be `iperf` TCP senders and two end-nodes to be `iperf` TCP receivers. In this configuration, LinkEM emulates four links with the desired bandwidth. We used bandwidths corresponding to typical modem connections, home DSL and cable modem connections, T1 lines, and 10 Mbps Ethernet connections. All tests were run for 120 seconds, and the `iperf` receivers measured the bandwidth every 5 seconds. Table 5.1 shows the results with the observed mean and percentage emulation error for all tests. We find that LinkEM accurately emulates link bandwidth within 2% of the configured bandwidth; for most tests it is within 1% of the target.

**Table 5.1**. Bandwidth emulation accuracy of LinkEM

| Configured bandwidth (Kbps) | Observed Mean bandwidth (Kbps) | Percentage Error |
|---|---|---|
| 64 | 63.46 | 0.84 |
| 128 | 126.51 | 1.16 |
| 256 | 254.06 | 0.75 |
| 512 | 508.12 | 0.75 |
| 1544 | 1531.62 | 0.80 |
| 2000 | 1984.79 | 0.76 |
| 10000 | 9920.84 | 0.79 |

### 5.1.2 Delay emulation

LinkEM's delay emulation accuracy was validated by sending `ping` packets of different sizes across it, and then using the round-trip time (RTT) to estimate the one-way emulated delays. We first measured the RTT between the end-nodes through LinkEM, with link delay configured to zero to get a base RTT value for different packet sizes. Table 5.2 shows the observed mean and standard deviation for the base RTT's. Then for each of those packet sizes, we again measured the RTTs with LinkEM configured to subject the packets to the desired latency. The difference gives an estimate of the accuracy of LinkEM's link latency emulation.

The topology for this experiment was the same as that for bandwidth emulation: four end-nodes connected through LinkEM, one node to each of its four ports. Link latencies were chosen to reflect typical latencies observed in the Internet: a few milliseconds to model latencies seen across local area networks on typical campuses, a few tens of milliseconds to model cross-country links, and about a

**Table 5.2**. Base RTT's for a range of packet sizes

| Packet Size (bytes) | Observed Mean RTT (ms) | Standard Deviation |
|---|---|---|
| 64 | 0.330 | 0.004 |
| 128 | 0.424 | 0.036 |
| 256 | 0.452 | 0.003 |
| 512 | 0.696 | 0.004 |
| 1024 | 1.058 | 0.020 |
| 1518 | 1.427 | 0.003 |

hundred milliseconds to model trans-atlantic links. Each test involved two nodes pinging the other two nodes by sending 10000 packets of a particular packet size, and then printing out the mean and standard deviation for the test. Tables 5.3, 5.4, and 5.5 present the RTTs for all tests for varying packet sizes, for one-way latencies of 5 ms, 30 ms and 101 ms respectively. The percentage errors across all tests show that LinkEM accurately emulates link delays.

**Table 5.3**. Delay emulation accuracy of LinkEM for configured delay of 5 ms

| Packet Size (bytes) | Observed Mean RTT (ms) | Standard Deviation | After sub. base RTT(ms) | Percentage Error |
|---|---|---|---|---|
| 64 | 10.317 | 0.028 | 9.987 | 0.13 |
| 128 | 10.407 | 0.044 | 9.983 | 0.17 |
| 256 | 10.459 | 0.007 | 10.007 | 0.07 |
| 512 | 10.703 | 0.007 | 10.007 | 0.07 |
| 1024 | 11.037 | 0.053 | 9.979 | 0.21 |
| 1518 | 11.434 | 0.012 | 10.007 | 0.07 |

**Table 5.4**. Delay emulation accuracy of LinkEM for configured delay of 30 ms

| Packet Size (bytes) | Observed Mean RTT (ms) | Standard Deviation | After sub. base RTT (ms) | Percentage Error |
|---|---|---|---|---|
| 64 | 60.366 | 0.014 | 60.006 | 0.01 |
| 128 | 60.487 | 0.017 | 60.063 | 0.105 |
| 256 | 60.491 | 0.009 | 60.039 | 0.065 |
| 512 | 60.734 | 0.008 | 60.038 | 0.063 |
| 1024 | 61.093 | 0.026 | 60.035 | 0.0583 |
| 1518 | 61.465 | 0.007 | 60.038 | 0.063 |

**Table 5.5**. Delay emulation accuracy of LinkEM for configured delay of 101 ms

| Packet Size (bytes) | Observed Mean RTT (ms) | Standard Deviation | After sub. base RTT (ms) | Percentage Error |
|---|---|---|---|---|
| 64 | 202.407 | 0.015 | 202.077 | 0.038 |
| 128 | 202.465 | 0.059 | 202.041 | 0.020 |
| 256 | 202.533 | 0.013 | 202.081 | 0.040 |
| 512 | 202.775 | 0.008 | 202.079 | 0.039 |
| 1024 | 203.107 | 0.053 | 202.049 | 0.024 |
| 1518 | 203.506 | 0.010 | 202.079 | 0.039 |

### 5.1.3 Loss rate emulation

LinkEM implements a uniform packet loss distribution model in which each link is configured with a fixed loss rate. Loss rate is measured as the ratio of packets dropped to the total number of packets sent. To validate the link loss accuracy, we used one-way traffic; the sender sends a fixed number of packets per link and the receiver reports the number of packets actually received. The difference gives the measured loss rate.

In this experiment, we used the OSKit [11] traffic generation and consumption kernels on end-nodes. The `send` kernel was configured to generate 120,000 maximum sized Ethernet packets at the rate of 8000 packets per second. The `consume` kernel simply received the packets, and printed out the statistics. Table 5.6 shows the observed loss rate and the percentage error from the configured rate. As seen from the table, LinkEM's loss rate emulation is accurate within 1% of the configured loss rate, except for one test in which the observed value is about 2.96% higher than the configured rate.

### 5.1.4 Summary of accuracy experiments

These experiments show that LinkEM's accuracy is comparable to that of Dummynet [36]. The bandwidth accuracy of both is within 2% of the configured bandwidth, the delay accuracy is within 1% of the target, while except for one outlier of 2.96% in LinkEM, both have loss rate accuracy within 1%.

**Table 5.6**. Loss rate emulation accuracy of LinkEM

| Configured link loss.(%) | Observed Mean loss rate (%) | Percentage Error |
|---|---|---|
| 1 | 1.0041 | 0.41 |
| 3 | 3.089 | 2.96 |
| 7 | 6.9983 | 0.02 |
| 10 | 10.0 | 0.0 |

## 5.2 Emulation Throughput

As discussed in Chapter 2, the packet forwarding capacity or throughput of a link emulator is largely independent of the packet size, much like that of a router. In this section, we report the measurement and comparison of LinkEM's and Dummynet's throughput for varying packet sizes. We used an 850 MHz PC with a 32-bit/33 MHz PCI bus as the Dummynet platform; this represented reasonably high-end PC hardware contemporary to when the IXP1200 was introduced. We used a polling FreeBSD kernel on the Dummynet node to avoid livelocks due to heavy interrupt load [24].

Figure 5.1 shows the topology for this experiment. Using Emulab's support for switched LANs, the link emulators (LinkEM or Dummynet) were connected in a topology with 16 end-nodes; 4 end-nodes were connected to each of the 4 100-Mbps ports of the emulator. Two of each set of four end-nodes ran the OSKit send kernels, while two ran the consume kernels. We needed two senders per port to generate line rate traffic for minimum-sized Ethernet packets. For all other packet sizes, a single OSKit sender was able to generate 100 Mbps line rate. The emulator was configured to emulate eight links in this configuration (corresponding to the eight send-consume traffic flows).

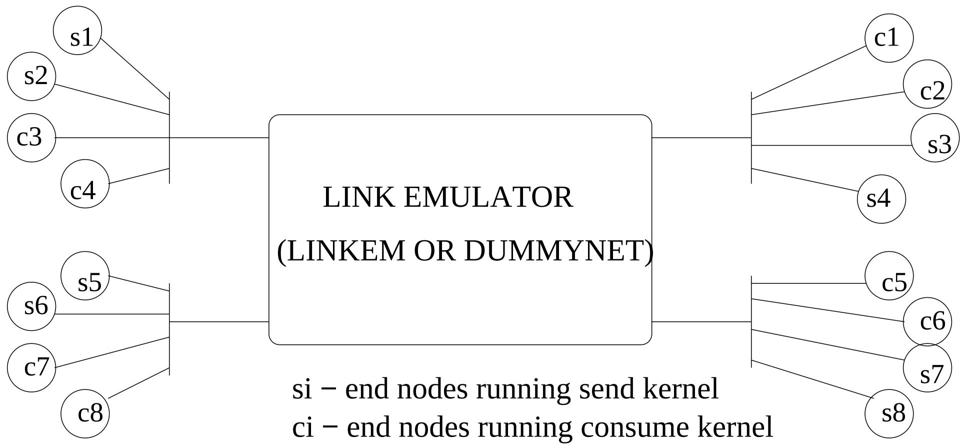


**Figure 5.1**. Throughput measurement experimental topology

All emulated links were configured at infinite bandwidth, zero delay, and zero packet loss rate to avoid any packet losses due to a link's characteristics. All measured packet losses in the consume kernels thus indicate an inability of the link emulator to keep up with the offered load. Infinite link bandwidth, zero delay, and zero loss rate for a link does not mean that the emulation task is bypassed in this configuration. Every packet is acted upon by the emulation task before being ejected out of the emulator. Thus this experiment measures the maximum throughput of both link emulators for different packet sizes.

### 5.2.1 LinkEM throughput

Figure 5.2 illustrates LinkEM's throughput for different packet sizes. The y-axis represents throughput as a percentage of the line rate for a particular packet size on the x-axis. The line rate in this experiment is 400 Mbps; the send kernels generate 100 Mbps traffic on each of the four ports. For minimum-sized packets (64 bytes) which is the worst case offered load, LinkEM can forward at around 83% of line

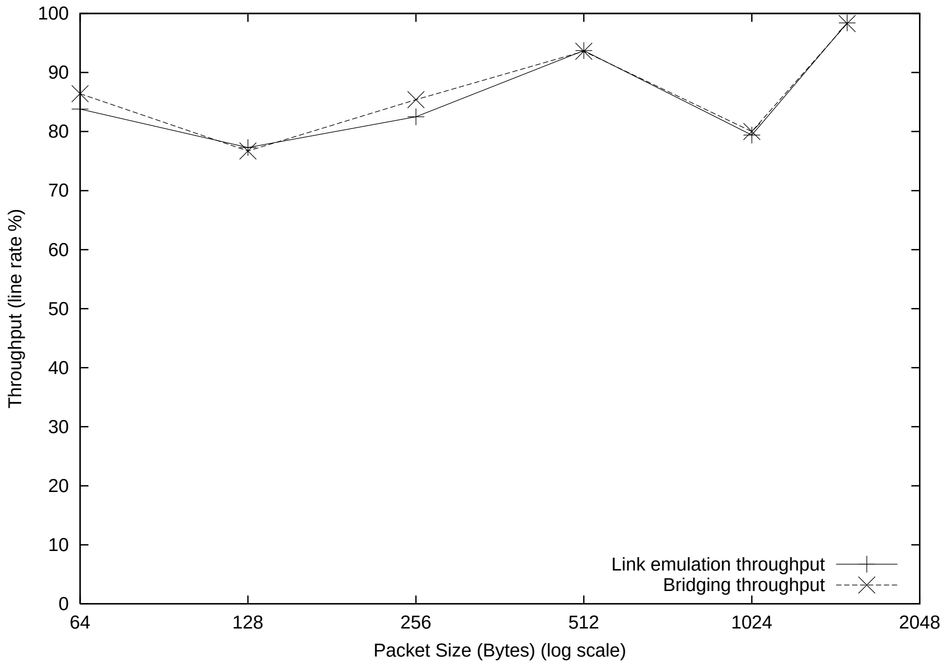


**Figure 5.2**. Throughput in percentage of line rate (400 Mbps)

rate or 332 Mbps (about 496 Kpps out of the offered 592 Kpps load). As the packet size increases, the throughput should approach 100% since the number of packets to be processed per second decreases.

However, Figure 5.2 shows two dips in the graph at packet sizes of 128 and 1024 bytes. Using packet traces, we discovered that the ingress and emulation microcode (Figure 4.14: Microblock Groups 1, 2, and 3) were able to forward at near line rate, however packets were dropped in the egress microcode (Figure 4.14: Microblock Group 4). Further code instrumentation revealed that the egress microcode was unable to move packets from memory to the output ports fast enough, which resulted in a number of packets being dropped at the output ports. As discussed in Figure 4.13, the egress microcode moves packets from the memory to ports in 64-byte chunks. The ports on the IXP1200 do not have enough buffer space to hold entire packets. Thus a port has to start transmitting a packet before the packet is fully assembled in its buffer. If the port starts transmitting a packet on the wire, and does not receive subsequent chunks fast enough, it simply appends an invalid CRC to the packet and gives up on transmission of the packet. Note that this is not a problem with 64-byte packets, since they have only one chunk. For packets greater than 64 bytes, which involve moving more than one chunk from memory to ports, the egress microcode is slow and results in underflow at the output ports.

To further verify that this was a problem in the egress microcode, and not in the emulation microcode or a side-effect of adding the emulation microcode, we ran only the Bridging reference code (without emulation tasks) provided by Intel, which includes the ingress, bridging, and the egress microcode. As seen in Figure 5.2, the graph for the Bridging reference code shows similar behavior.

Figure 5.3 shows the throughput of LinkEM and the Bridging reference code measured just before packets enter the egress microcode (thus all egress drops are ignored). As seen in the graph, LinkEM forwards at line rate for all packet sizes other than 64 bytes. In Appendix A, we analyze the current one-microengine egress design. We then sketch a design for the egress task that uses two microengines to support higher transmission rate. The sixth idle microengine in our current design

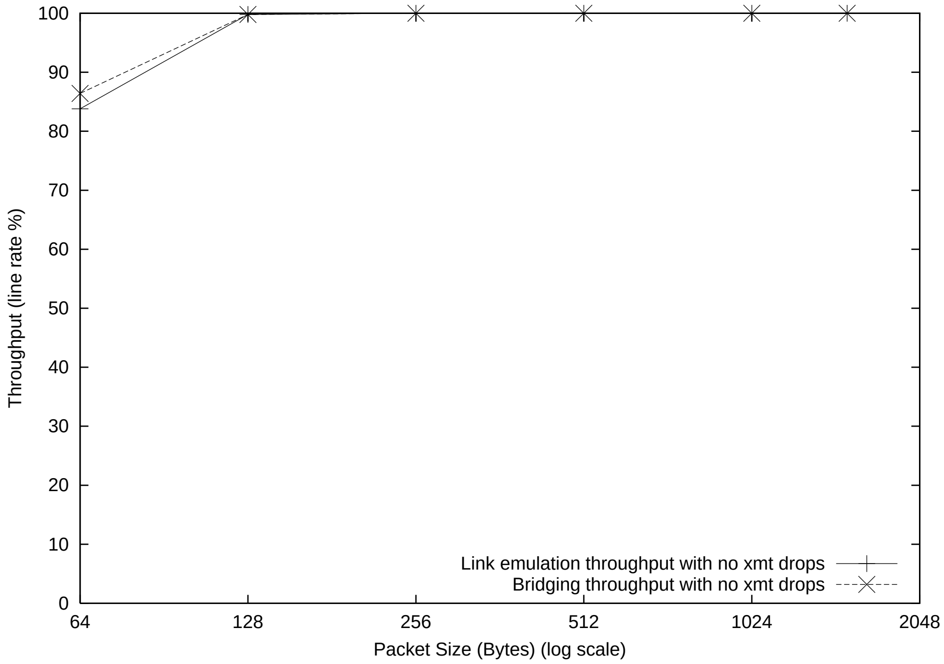


**Figure 5.3**. Throughput in percentage of line rate (400 Mbps), no drops in egress

can be assigned to speed up egress. We leave the implementation of the new egress design as future work.

### 5.2.2 Dummynet throughput

Figure 5.4 shows Dummynet throughput across different packet sizes for a line rate of 400 Mbps. For minimum-sized packets, Dummynet can forward at about 18% of line rate or 72 Mbps (106 Kpps out of the offered load of 595 Kpps). For maximum-sized packets, the throughput is 81% of line rate or 324 Mbps (about 26.2 Kpps out of the offered load of 32 Kpps). As seen in the figure, LinkEM has a higher throughput of between 4.6 and 1.2 times that of Dummynet over the range of packet sizes.

The 32-bit/33 MHz PCI bus becomes a bottleneck when large packet traffic is sent across all four ports of the Dummynet node. To measure Dummynet's forwarding capacity in the absence of a PCI bottleneck, we used the same experimental topology, but sent in traffic across only two ports. This corresponded to a line rate of 200 Mbps.

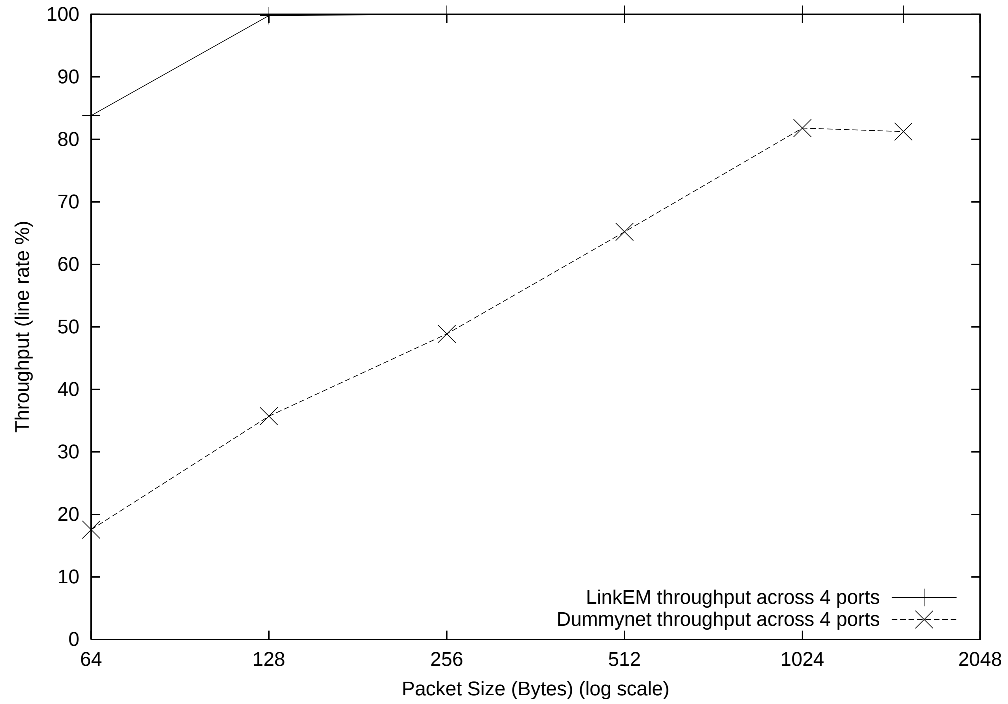


**Figure 5.4**. LinkEM and Dummynet throughput in percent line rate (400 Mbps)

Figure 5.5 shows the throughput seen for different packet sizes as a percentage of line rate (200 Mbps). As seen in the figure, Dummynet can keep up with near line rate for packet sizes at and above 256 bytes. For minimum-sized packets, the throughput is about 42% of line rate or 84 Mbps (about 124 Kpps of the offered load of 296 Kpps). For 128-byte packets, the throughput is 62% of line rate or 124 Mbps (about 104 Kpps out of the offered load of 168 Kpps). Thus LinkEM has a higher throughput of 2 and 1.6 times that of Dummynet for these packet sizes respectively. For packet sizes greater than 128 bytes, both can forward at line rate in this configuration.

## 5.3 Link Multiplexing Capacity

As described earlier, link multiplexing is a useful feature of a link emulator. It allows an emulator to emulate multiple lower-speed links over a single physical link. For instance, one can emulate a number of slower network links like cable, DSL, T1

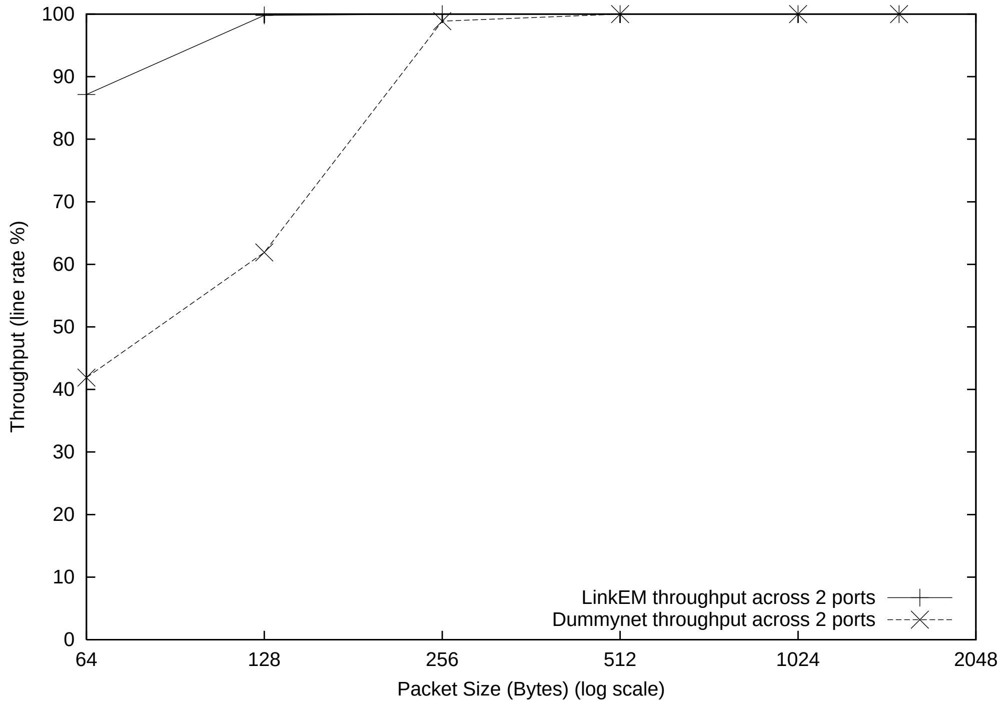


**Figure 5.5**. LinkEM and Dummynet throughput in percent line rate (200 Mbps)

or 10 Mbps Ethernet on a single 100 Mbps physical link. LinkEM identifies links by a 64-bit key consisting of the source and destination IP address in a packet. A packet is classified early on to its link, and then subjected to the link's characteristics. Like Dummynet, LinkEM uses a heap to implement a priority queue data structure of active link queues. The key or priority is the expiration emulation time for the first packet in a queue. Each tick, all active link queues that expire are handled by the emulation task. This section describes an evaluation and comparison of link multiplexing capacity of LinkEM, and Dummynet on a 850 MHz PC.

### 5.3.1 Experiment setup

To measure the multiplexing capacity, we had to generate a large number of traffic flows with different source and destination IP addresses. Each such flow can be thought of as a link. We used TCP flows since they are responsive to the available link bandwidth and can be used as a measure of how well the bandwidth of a link is constrained by a link emulator to a configured value.

We use a mechanism provided in FreeBSD, called `jail` [21], to generate a large number of end-node TCP flows with different source-destination IP addresses. The `jail` mechanism is used to partition the operating system into multiple virtual machines, with a process and its descendents jailed into a particular virtual machine. Multiple jails coexist on a single physical node, and can be configured with different IP addresses, with restricted access to files within the jail. A pair of jails on two end-nodes can be used as a source and destination node for a TCP flow, thus giving us the ability to generate a large number of TCP flows with a relatively small number of physical machines.

### 5.3.2 Link multiplexing

The same `jail` mechanism is used to generate traffic with either LinkEM or Dummynet configured in between as the link emulator node. We configure the links inside LinkEM or Dummynet to the desired bandwidth, and run TCP flows between the end-nodes sending traffic through the link emulator. Each TCP flow is run for 120 seconds, and every TCP receiver measures the link bandwidth at 5-second intervals. If the bandwidth perceived by the flow is within accuracy limits[1] of the configured bandwidth, then we know that the emulator can handle that many flows. We increase the number of flows until the emulator cannot keep up with the number of multiplexed links or we reach the maximum number of multiplexed links that can fit on to a physical link.[2] Note that the number of multiplexed links is twice the number of flows, since each pair of end-node IP addresses is treated as two separate links so that asymmetric links can be supported. For instance, the forward link can

---

[1] We use 2%, since that is the bandwidth accuracy limit when the system is not loaded, see Table 5.1

[2] For instance, if a single TCP flow can get a maximum of 94 Mbps through the link emulator, then we can multiplex about 47 2-Mbps flows on that physical link.

be configured to 128 Kbps while the reverse link can be independently configured to 256 Kbps to model a DSL-type asymmetric link.

Figure 5.6 depicts the link multiplexing capacity of both LinkEM and Dummynet. The x-axis represents the multiplexed link bandwidth which ranges from 128 Kbps to 10 Mbps and is plotted in logarithmic scale. The y-axis represents the number of multiplexed links supported by the emulators at a particular bandwidth. The third curve in the figure, the ideal link multiplexing capacity, shows the number of multiplexed links that can fit into a physical link at a particular bandwidth. We expect that both the emulators will follow the ideal curve at bandwidths close to 10 Mbps, since the number of links that need to be emulated at this bandwidth is small, and will diverge from the ideal curve at lower bandwidths (closer to 128 Kbps), as they will be unable to keep up with the high number of multiplexed links without losing accuracy.

In this experiment, the four ports of the emulator are divided into two port pairs, and lower bandwidth links are multiplexed on each port pair. Thus a point

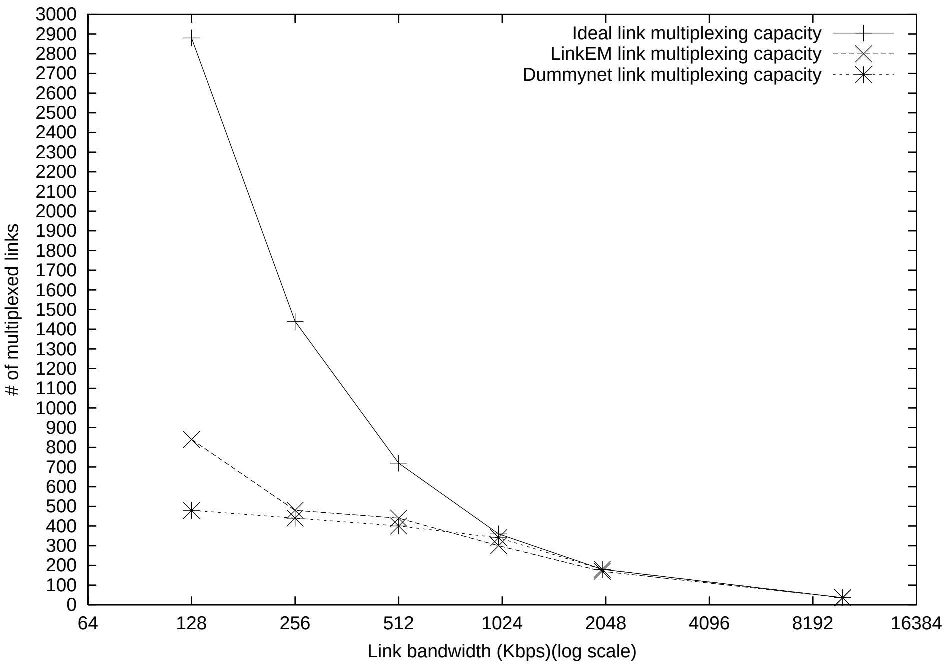


**Figure 5.6**. Link multiplexing with traffic across all four ports

(x,y) in the graph implies that y multiplexed links, each of x Kbps bandwidth, are supported by the emulator, with y/2 links multiplexed on each of the two port pairs. Figure 5.7 illustrates the topology for this experiment.

Figure 5.6 shows that Dummynet can support about 36 10-Mbps links (18 TCP flows, 9 flows across each pair of ports), and the number of links increases to around 480 (240 TCP flows, 120 flows across each port pair) at a link bandwidth of 128 Kbps. For every test, Dummynet can accurately emulate the configured bandwidth up to the number of links shown in the graph, and as the number of links is further increased, the accuracy starts dropping as it cannot keep up with the offered emulation load. The Dummynet link multiplexing curve coincides with the ideal curve between 10 Mbps and 2 Mbps bandwidths, and diverges from the ideal curve to a lower number of multiplexed links at bandwidths of 1 Mbps and below.

From the figure, we see that LinkEM can support about 36 10-Mbps links (18 TCP flows, 9 flows across each pair of ports), and the number of links increases to about 840 (420 TCP flows, 210 flows across each port pair) at a link bandwidth of 128 Kbps. The LinkEM link multiplexing curve is close to the ideal curve between 10 Mbps and 2 Mbps bandwidths, and only diverges from the ideal curve at bandwidths of 1 Mbps and below. Compared to Dummynet, LinkEM is able

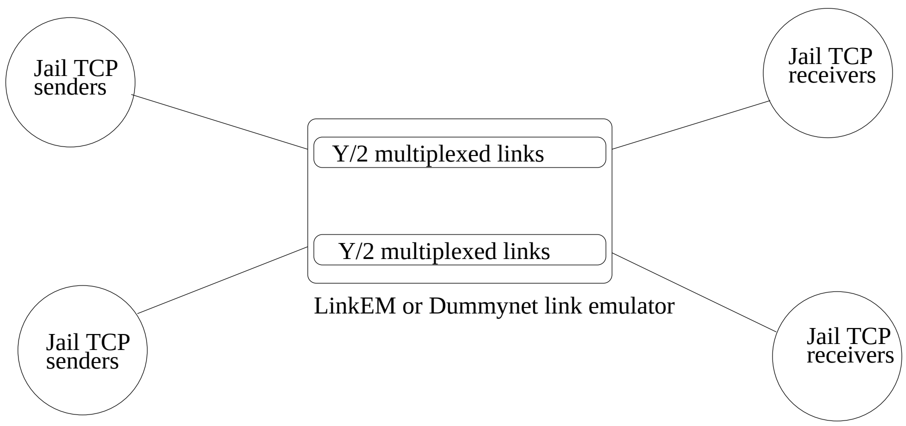


**Figure 5.7**. Link multiplexing topology with traffic across all four ports

to multiplex almost the same number of links at high bandwidths (36 vs 36 at 10 Mbps, 172 vs 180 at 2 Mbps), and can emulate more links at low bandwidths (840 vs 480). However, LinkEM's multiplexing capacity is plagued by the same egress microcode drops which we described in Section 5.2. Port underflows cause some TCP flows to back off, thus resulting in inaccurate bandwidth emulation for those flows. This behavior is observed for the entire range of multiplexed link bandwidths from 2 Mbps to 128 Kbps.

We then slightly modified the experiment to run traffic flows across only one pair of ports, to avoid the drops in the egress microcode. The goal was to try to reach the emulation bottleneck before reaching the egress microcode's limit. Figure 5.8 depicts the multiplexing capacity of LinkEM and Dummynet in comparison with the ideal multiplexing curve, with traffic across only two of the four ports of the emulators. Thus a point (x,y) in the graph implies that y virtual links, each of x Kbps bandwidth, are supported by the emulator, with all y virtual links multiplexed on a single port pair. Figure 5.9 illustrates the topology for this experiment.

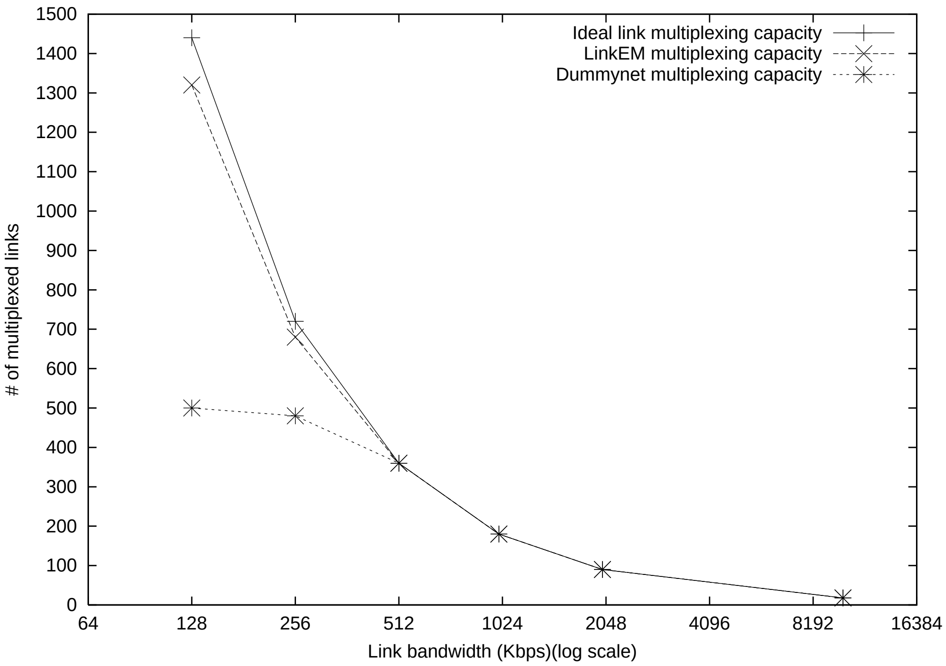


**Figure 5.8**. LinkEM multiplexing with traffic across only two ports

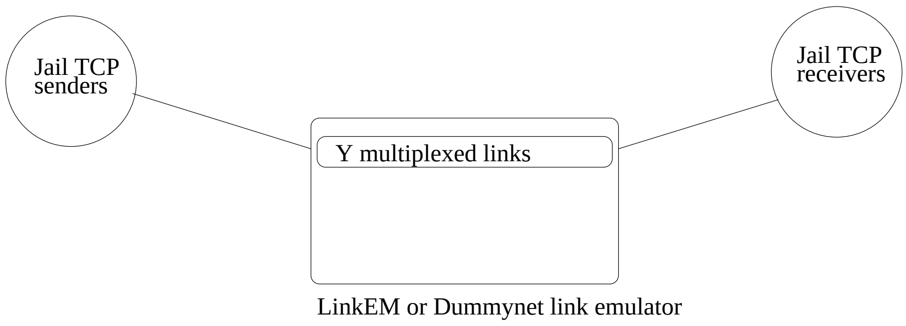


**Figure 5.9**. Link multiplexing topology with traffic across two ports

As seen in the figure, the Dummynet curve coincides with the ideal curve between 10 Mbps and 512 Kbps bandwidths, and diverges from the ideal curve to lower number of multiplexed links at 256 Kbps and 128 Kbps link bandwidth. At these bandwidths, Dummynet can accurately emulate about 500 links (250 flows), or about 66% (at 256 Kbps) and 34% (at 128 Kbps) of the ideal number of multiplexed links. The LinkEM curve coincides with the ideal curve between 10 Mbps and 512 Kbps bandwidths, and diverges from the ideal curve to about 680 links (340 flows) and 1320 links (660 flows), or to 95% and 90% of the ideal number of links at 256 Kbps and 128 Kbps respectively. This experiment shows that LinkEM's multiplexing capacity is higher than Dummynet on our experimental PC platform by between a factor of 1.4 (95% vs 66%) and 2.6 (90% vs 34%) for multiplexed links of low bandwidth. For multiplexed link bandwidths of 512 Kbps and higher, both are able to match the ideal curve.

# CHAPTER 6

# RELATED WORK

Hitbox[1] is probably the earliest PC-based in-kernel link emulator. It was primarily developed to study and compare TCP Vegas [4] with TCP Reno [19] outside of simulation. Dummynet [30], NIST Net [25], and ONE [2] are single-node PC-based link emulators which provide Hitbox-like in-kernel link emulation. The emulation module sits in between the stack and the network interface. It intercepts traffic and emulates link characteristics, completely transparent to the user-level application. These emulators can run inside the end-node kernels, or can run on interposed PCs. LinkEM's emulation model is based on Dummynet; however, it is implemented on a specialized network processor platform instead of a PC.

Modelnet [34] is a large-scale network emulation environment which uses a Dummynet-like implementation to provide link emulation. A Modelnet core node (link emulator node) is a complete reimplementation of Dummynet on a FreeBSD PC, and in addition, supports hop-by-hop link emulation. The path between two end-nodes in the topology is distilled to a set of links, which are then emulated inside one or more core nodes. Similar to a Modelnet core node, LinkEM can also be extended to support hop-by-hop emulation.

End-to-End Delay Emulator (ENDE) [37] measures the end-to-end characteristics between two nodes on the Internet, and then emulates the link inside a Linux-based link emulator. The path characteristics are measured using packet-pair probe techniques [22] and using one-way latency measurement techniques [29, 28], and are fed into the emulator. ENDE thus combines measurement of the characteristics of a single link and its emulation into a single tool. LinkEM does not support measuring a link's characteristics, but assumes that experimenters use other tools for that purpose, and then feed the link characteristics into LinkEM.

Trace Modulation [27] is a technique for emulating mobile wireless networks on a wired network. The wireless application is first run in a real wireless topology and packet traces are collected. Parameters of the wireless link are derived based on a model and the traces, and these characteristics are then emulated on a wired network by using Hitbox-style in-kernel link emulation.

NS Emulation (NSE) [9] supports an environment in which packets from the real world are injected into a simulation and vice versa. One of the main advantages of NSE is that one can study the interaction of real traffic with the rich and well tested background traffic models running as part of the simulation. However, since NSE runs in a single process in user-space, its emulation capacity is not as high as in-kernel link emulators.

Network Emulation Tool (NET) [3] is an emulator for distributed systems that emulates a site of nodes connected by a network that loses, duplicates and delays messages. A node at each site in the virtual topology is mapped to a different unix process. Applications are bound to a library that implements the send and receive system calls, and diverts all messages to a central process, which emulates the underlying network. Delayline [13] uses a similar model wherein applications bind to a library that emulates the network. Thus both these approaches are not transparent to the application, and require a fair amount of work to provide the same semantics to the applications as those provided by the real network library.

Network simulation has link simulation as one of its components. Link simulation is essentially similar to link emulation, except that unlike emulation, it does not have to meet real-time requirements. Thus a link simulator can potentially simulate any number of links or links of any speed. The Harvard TCP/IP simulator [35] uses the host kernel's TCP/IP stack to simulate end-nodes and routers, and special link objects in user-space to simulate the links. Tunnel interfaces are used to move packets in between the kernel part of the simulator and the user-level link simulator. The Entrapid Protocol Development Environment [12] virtualizes the BSD networking stack and moves it to user-space. Each such instance of a stack forms a virtual node in the simulation. Virtual nodes are connected by link simulation modules which can simulate point-to-point and Ethernet links.

The IXP network processors use parallelism, multithreading and hardware assists to support packet processing at high line rates. However, it is a significant challenge to map an application to take advantage of this parallelism while avoiding communication and synchronization bottlenecks. The Network Systems group at Princeton has evaluated the IXP1200 network processor for building a robust, high capacity, and extensible router [32]. The design approach that they employ is to split the application into two parts: a Router Infrastructure (RI) part that handles vanilla IP packet forwarding and a Virtual Router Processor (VRP) part that handles additional processing per packet. Based on the line rate that needs to be handled, and the processing cost of the RI, the available budget for the VRP is calculated. This budget is then used to add additional processing per packet in the fast path.

The software programming model [20] on the IXP2400 and IXP2800 augments the MicroACE programming framework with design techniques to map an application on the 8 and 16 microengines, and 64 and 128 threads respectively on these processors. This model includes Context pipeline stages, Functional pipeline stages, a Pool of Threads model (POTS) to support out-of-order execution, and strict sequencing of threads to support in-order execution. A pipeline of context stages is similar to our Microengine pipelining design method, while the functional pipeline stages are similar to the Microengine parallelism technique. The POTS and in-order thread execution techniques are two types of functional pipeline stages that support applications with different kinds of packet processing requirements. See [20] for more details.

The IXA SDK-3.0 that Intel will supply with the next generation IXP network processors has a new programming model called the auto-partitioning programming model [14]. This model automates mapping of an application to the parallel resources of the network processor using the design techniques of context and functional pipeline stages, and POTS and ordered thread execution. Programmers specify their application as a sequence of packet processing stages (PPSes) that can execute in parallel. Each stage is written in C and augmented with constructs, using which the programmer can specify throughput requirements of each PPS and can identify critical sections of code. The C compiler then automatically maps PPSes to multiple microengines and threads to meet the throughput requirements. Thus, this programming model maps straight-line C code to the parallel microengines while automatically providing communication and synchronization primitives.

Netbind [5] is a binding tool for creating new datapaths in network processor based routers. While the MicroACE architecture [16] statically binds microcode components using a dispatch loop, Netbind supports binding of components during runtime by modifying entry and exit points of components directly in the component binary. The paper shows that this binding method has lower overhead than the MicroACE binding method; however the programming model exposed is more restrictive than the MicroACE programming model. The register space of a microengine is statically divided, and components have to fit their data within the allocated space, or have to spill data to memory.

The Active System Area Networks (ASAN) group at Georgia Tech. has evaluated the use of the IXP1200 as a smart network interface on a host PC [23]. The IXP1200 exports an ethernet interface to cluster computing applications on the host. The paper reports bandwidth and latency measurements for transferring messages of different packet sizes across the PCI bus in both directions, from the host to the IXP and from the IXP to the host, and estimates the headroom available for computation on the microengines, in addition to the cycles spent in the transfer.

# CHAPTER 7

# DISCUSSION AND CONCLUSIONS

Existing solutions for link emulation have used general-purpose PC-class machines. The well-understood hardware and software PC platform make it ideal for quick implementation and deployment. Until now, PCs have faithfully tracked Moore's law, resulting in increasing performance at decreasing costs for most applications. As discussed in Chapter 1, the PC architecture is largely optimized to exploit two traits in applications: large amounts of ILP and good memory reference locality. However, this approach has begun to have diminishing returns, even for applications that demonstrate these traits. The returns are likely to be even less for networking applications, which in general have little ILP [7] and only moderately good locality characteristics [6]. As networking applications scale to support larger packet rates and higher bandwidths, this mismatch is likely to become more evident. The demand for higher aggregate emulation bandwidths (multi-gigabit) and for emulating faster network links,[1] might require a different approach to link emulation than just deploying faster PCs.

An alternative is to use specialized platforms optimized for network processing, for link emulation. Network processors are programmable processors that use a multithreaded, multiprocessor architecture to exploit packet-level parallelism and have instruction sets and hardware assists optimized for common networking tasks. In this thesis, we have evaluated the IXP1200 network processor for building a high capacity link emulator, LinkEM, and have compared it with Dummynet running on a PC hardware configuration contemporary with the IXP1200. On this platform, LinkEM has between a factor of 1.6 and 4.6 higher throughput for small-sized

[1] for instance OC-12 links at 622 Mbps

packets while both are able to forward at near line rate for large packets. LinkEM's link multiplexing capacity was between 1.4 and 2.6 times higher than Dummynet at bandwidths below 256 Kbps while both were able to emulate the maximum number of multiplexed links at bandwidths above 512 Kbps.

Moore's law applied to network processors implies that in the future more and more multithreaded packet processing engines can be packed into the chip, and at the same time, network processing specific hardware support can also be integrated closer to the packet processing engines, resulting in increasing performance at decreasing costs. For instance, the first generation Intel IXP1200 network processor has six microengines, an on-chip hardware hash unit, low latency on-chip hardware synchronization primitives, and is targeted to data rates of 155 Mbps to 600 Mbps with a per chip price of around $200 in low volumes. The second generation processors, the IXP2400 [17] and IXP2800 [18], pack between 8 and 16 microengines per chip, with a higher level of multithreading, have on-chip hardware support for atomic queuing, a small amount of on-chip local memory per microengine, a small content addressable memory (CAM), and are targeted at 2.5 Gbps to 5 Gbps data rates with chip prices between $230 to $500 in low volume.

This trend will likely continue, as these chips are used increasingly often by OEM manufacturers to build networking equipment. Unfortunately, generic development boards built around these chips are still expensive, due to the low volume. The ENP-2505 board [8] built using the IXP1200 we used, costs about $2000 in low volume, up to 10 times the IXP1200 chip cost. Thus, while it is possible to build high-capacity link emulators on network processors, at this point, they do not seem to offer a better price-performance alternative to PCs for link emulation.

It is interesting to note that link emulation processing is very similar to traffic shaping at edge routers. Link emulation involves demultiplexing an incoming packet stream into its constituent emulated links, and subjecting packets to the configured link bandwidth, delay and loss rate characteristics. Traffic shapers similarly classify packets into different flows, and then restrict flows to the preset service level agreements. They are deployed by ISPs at the edge of the network to police traffic entering the core and ensure that it is compliant with the user's subscription level. Service differentiation is a big financial incentive for ISPs and therefore also for edge router OEM manufacturers. Both of them would like to build devices that can be rapidly programmed and upgraded to support new service models without resorting to expensive hardware redesign cycles. Since network processors tout flexibility through programmability, traffic shaping is a promising good fit application for this technology. An indirect effect of network processors being used in large volumes for traffic shaping applications would be better price-performance returns for using them for link emulation, as the cost for development boards for these applications goes down.

In addition to hardware costs, there is a software development and maintenance cost that should also be factored in. The one-time software development cost of a link emulator on a network processor is higher than development of an in-kernel implementation on a PC, mainly due to the challenge of programming in a parallel environment. A PC platform is well-understood, has a large number of familiar compilation, debugging and optimization tools, and permits a high degree of software reuse across different generations of PC hardware, thereby reducing the overall programming and maintenance costs. On the other hand, software development platforms for network processors are still not as mature as their PC counterparts. It is a significant challenge for a programmer to map the tasks of an application (in this thesis a link emulator), across the multiple microengines and threads of a network processor in order to extract maximum performance. Hence, network processor vendors are putting efforts into development of design tools and compilers which abstract the hardware without incurring heavy performance hits, and on programming models which support reusability of code across different generations of the same network processor [16]. As these tools mature, we believe that the overall software development cost will reduce and will be tolerable.

The current generation high-end PCs are clocked at between 2 and 3 GHz, have large amounts of on-chip cache, and have a high bandwidth PCI-X or 64-bit/66 MHz PCI bus. As a comparison point, the IXP2400 [17], one of the processors in the current generation of Intel network processors, has 8 microengines clocked at 600 MHz each, 8 hardware threads on each microengine, and other features like per microengine local memory and CAM, and on-chip support for fast packet queuing. While the IXP2400 is richer in resources, the basic multithreaded, multiprocessor architecture remains the same as the IXP1200; the lessons which we learned while implementing LinkEM on IXP1200, can be applied while porting LinkEM to the new processor. It will be interesting to compare Dummynet on current PC hardware against LinkEM on current network processor hardware.

# APPENDIX

# EGRESS TASK DESIGN

## A.1 Transmit State Machine Basics

The Transmit State Machine (TSM) is located inside the on-chip IX bus interface, and along with the transmit microengines is responsible for moving packets out of the IXP1200. Figure A.1 shows the different hardware components involved in packet transmission.

The IX bus interface consists of 16 64-byte buffers (TFIFO slots) which are treated by the TSM and the microengines as a circular queue. Packets are trans-

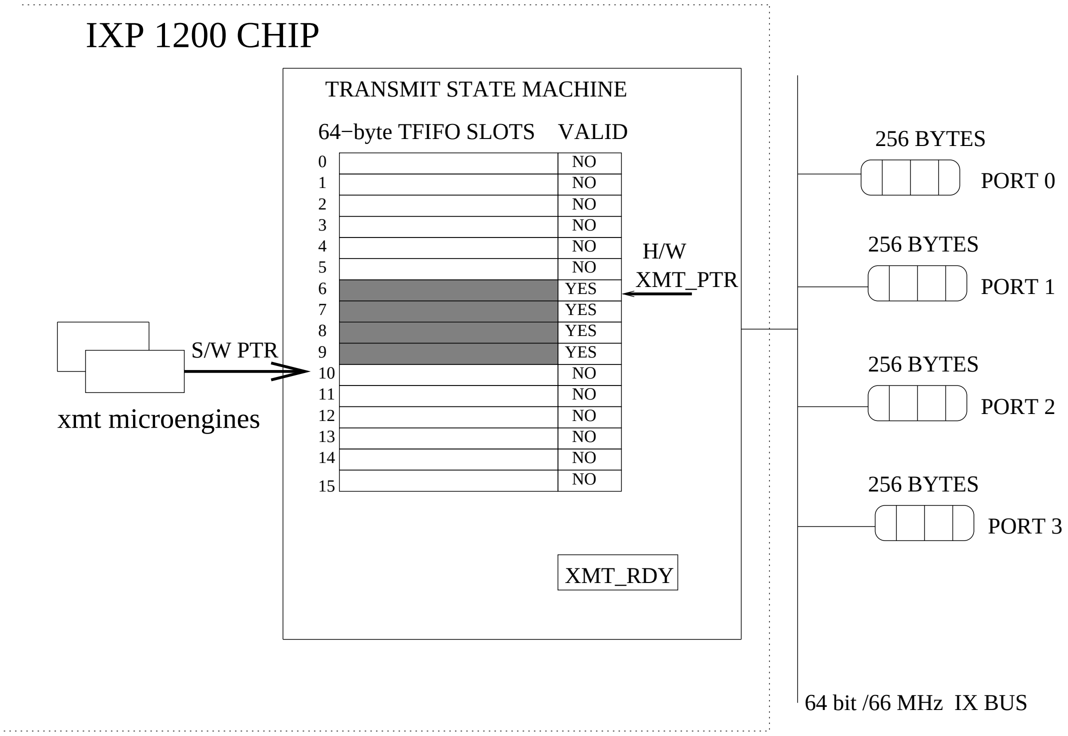


**Figure A.1**. Packet transmission on the IXP1200 ferred from memory to output ports in 64-byte chunks, and the TFIFO slots are used to stage these chunks during the transfer. Each TFIFO slot has an associated valid bit and some control bytes which have information about the port the chunk is supposed to be sent to, and the position of the chunk in the entire packet (whether start of packet, end of packet, or some intermediate chunk). Once a chunk is moved to a TFIFO slot, the transmit microengines set up the control bytes and the valid bit associated with the chunk. Valid chunks are moved by the TSM from TFIFOs to port buffers over the IX bus. The TSM maintains a hardware pointer (H/W XMT_PTR) to the current TFIFO slot that it is processing. Similarly, the microengine maintains a pointer (S/W PTR) to the TFIFO slot which is being filled up from memory. The IX bus interface has a status register called XMT_RDY which holds information about the amount of free buffer space in the port.[1]

The transmit microengines have four main responsibilities to ensure correct packet transmission:

- **Avoid garbage data from being sent to the ports:** This is done by making sure that the TSM's hardware pointer never crosses the microengine software pointer. The microengine sets the valid bit only for valid chunks, and the TSM processes only valid chunks in order, invalidating them after processing. If a chunk is not valid, the TSM stalls waiting for the chunk to become valid before proceeding ahead.

- **Avoid overwriting data before it is sent to the port:** This is done by making sure that the microengine software pointer does not cross the TSM's hardware pointer. The microengine can peek at the TSM's hardware pointer through a status register.

- **Avoid port overflows by not filling and validating the TFIFOs faster than the rate at which the port can transmit:** Each port has a buffer of 256 bytes, and thus cannot hold more than four 64-byte chunks. Moving

[1] Each port has 256 bytes of buffer space and can hold up to four 64-byte packet chunks.

data from the TFIFO slots to the port faster than the port can transmit it on the wire results in port overflows. The amount of available space in a port's buffer is reflected in the XMT_RDY status register. The transmit microengines check this status and validate the TFIFO slots only when there is space in the port buffer. This essentially throttles the rate at which the TSM moves packets to the ports and thus avoids port overflows.

- **Avoid port underflows by filling the port as fast as it can transmit:** Each port has a buffer of 256 bytes; thus for large packets, the port has to start transmitting the packet on the wire before the entire packet is transferred from the TFIFOs to the ports. This means that the TSM and the microengines have to move data from memory to the ports, through TFIFO slots, at line rate, otherwise, the port buffers will underflow resulting in packet transmission errors. This is the problem with the current one-microengine Egress design (see section 5.2).

  The 64-bit/66 MHz IX bus supports data transfers up to 4.2 Gbps. Thus this capacity is enough to support moving packets from the TFIFO slots to ports at line rate on our board (four 100-Mbps ports). Thus the bottleneck is the rate at which the microengines can move packets from memory to the TFIFO slots. In the new Egress design described in section A.3, we devote two microengines to the Egress task to avoid this bottleneck. Shared resources like TFIFO slots, transmit queues, and transmit ports are statically allocated to the two microengines to avoid synchronization costs.

## A.2 The Current One-microengine Egress Design

This section describes the current design of the egress task as implemented in Intel's reference design microcode which we have used in LinkEM. The egress task dequeues buffer handles and other meta information from queues in memory and transfers the packets to output ports for transmission using the IX Bus Interface. Thus it consists of two logical subtasks: scheduling or selecting the next packet from a set of queues (scheduler), and moving the packet from memory to TFIFO for transmission (fill). Since the scheduler subtask in egress executes simple round-robin algorithm over all transmit queues, it does not require much processing. The fill task however is highly memory and IO bound as it is responsible for the actual packet transfer, and also slightly more involved, since it interacts with the TSM for maintaining the four conditions described in section A.1.

One thread (scheduler thread) on the microengine is used to execute the scheduler subtask; it executes round-robin scheduling on all transmit queues and stores the dequeued buffer state in per-port microengine registers. The other three threads (fill threads) use this buffer state and move the packet from memory to TFIFO slots for transmission. Since there are three fill threads, they are mapped to every third TFIFO slot. The 16 slots are statically mapped to the four ports (four slots per port). Each fill thread computes its next slot, computes the port corresponding to that slot, and moves the next packet chunk destined for the port to the TFIFO slot. It then validates the chunk while maintaining the conditions outlined in section A.1. Figure A.2 shows the mapping of TFIFO slots to ports and fill threads.

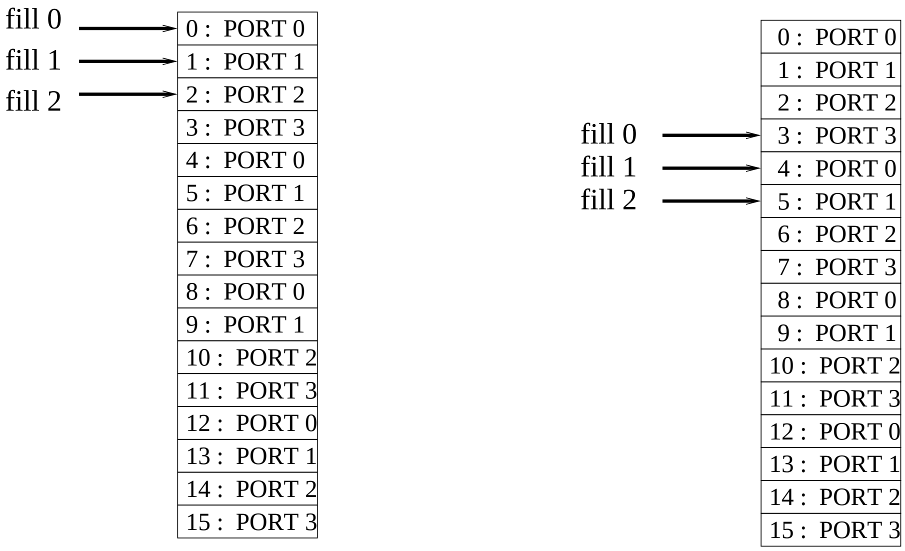


**Figure A.2**. TFIFO to port mapping

### A.2.1 Packet transmission on all four ports

Figure A.3 shows a snapshot of packet transmission in this design on all four ports. Let's assume that fill threads 0, 1 and 2 have moved the $i^{th}$ chunk of a packet destined to port 0 to TFIFO 0, $j^{th}$ chunk of a packet destined to port 1 to TFIFO 1, and $k^{th}$ chunk of a packet destined to port 2 to TFIFO 2 respectively. Then, assuming that the ports have space for these chunks in their buffers, the three threads validate the TFIFO slots so that the TSM can move them to the ports. The threads then increment their TFIFOs by 3, and move the $m^{th}$ chunk of a packet destined to port 3, $i+1^{st}$ chunk of a packet destined to port 0, and $j+1^{st}$ chunk of a packet destined to port 1, from memory to the corresponding TFIFO slots respectively. Thus the $i^{th}$ and $i+1^{st}$ chunks of a packet destined to port 0 are transferred one by one from memory to TFIFO slots. If the fill threads are not fast enough to do this transfer at line rate for port 0, then the TSM hardware pointer will be stalled at TFIFO slot 4 waiting for the $i+1^{st}$ chunk to be transferred from memory and validated, thus resulting in an underflow on port 0.

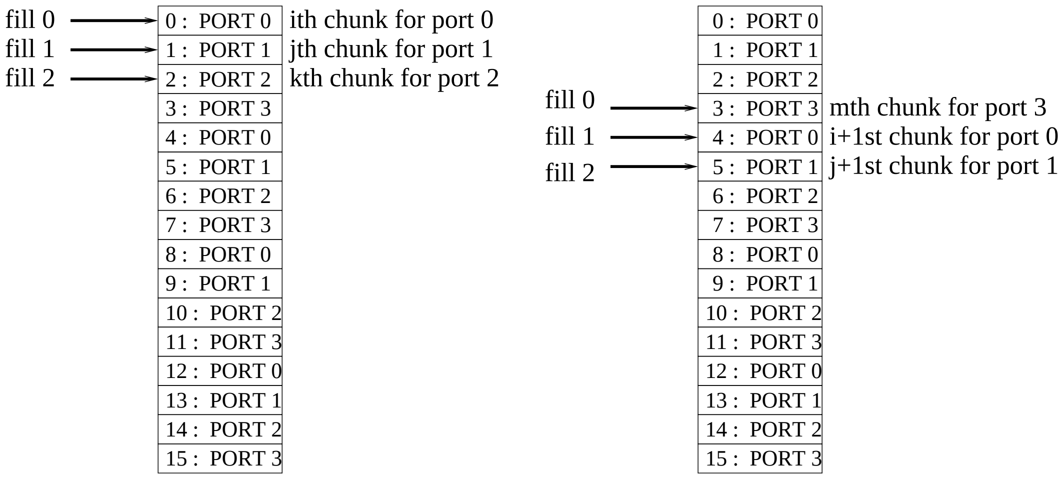


**Figure A.3**. TFIFO filling with transmission on all four ports

### A.2.2 Packet transmission on only two of the four ports

Let's assume now that packets are being transmitted only on two of the four ports (ports 0 and 2). If a fill thread is processing a TFIFO slot which corresponds to a port that does not have a packet for transmission, it sets an error bit in the control bytes associated with the slot and validates it. It then increments its slot number by 3 and starts processing the new slot. The TSM on coming across a TFIFO slot with the error and valid bit set, increments its hardware pointer beyond the slot, but does not transmit the chunk in that slot to any port. Figure A.4 shows a scenario where ports 0 and 2 have data for transmission, while ports 1 and 3 do not have any data. An "X" next to a slot in the figure implies that the fill thread has set the error bit for that TFIFO slot since the corresponding port does not have any packet for transmission.

As seen in the first column of the figure, fill thread 0 is processing the $i^{th}$ chunk of a packet destined for port 0, while fill thread 2 is processing $j^{th}$ chunk of a packet destined for port 2. Since there is no packet for port 1, fill thread 1 skips TFIFO element 1, jumps 3 slots ahead to TFIFO slot 4 and starts processing $i+1^{st}$ chunk of the packet for port 0. Thus two consecutive chunks of a packet are being transferred from memory to TFIFO slots simultaneously.

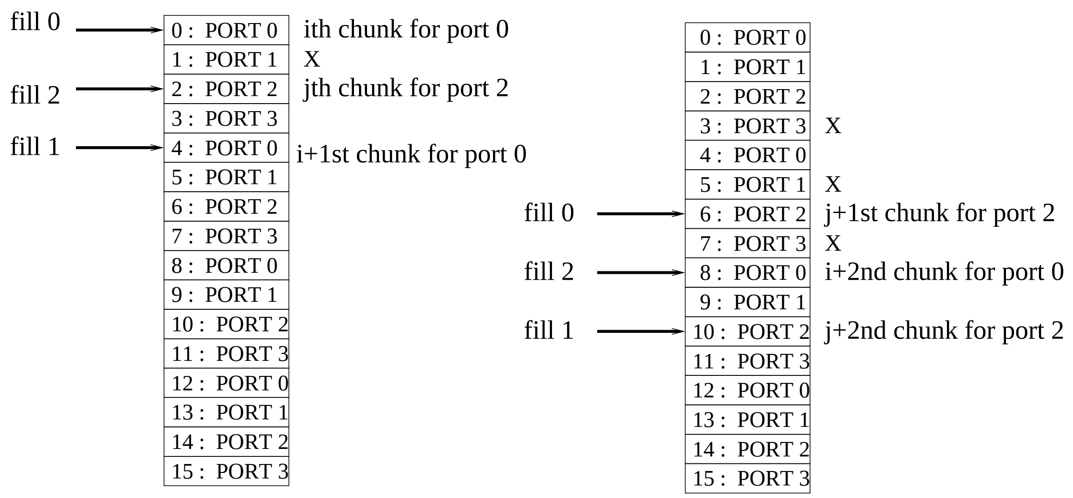


**Figure A.4**. TFIFO filling with transmission on only ports 0 and 2

Once the threads move the current packet chunks and validate them, they each increment their TFIFO slots by 3. Fill thread 0 skips TFIFO slot 3 (since port 3 does not have any data), and then starts transferring the $j+1^{st}$ chunk of the packet destined for port 2 to slot 6. Similarly, fill thread 2 skips TFIFO slot 5 (since port 1 does not have any data), and starts transferring $i+2^{nd}$ chunk of the packet destined for port 0 to slot 8. Fill thread 1 skips TFIFO 7 and starts transferring $j+2^{nd}$ chunk of the packet destined for port 2 to slot 10. Two consecutive chunks of a packet destined for port 2 are being moved from memory to TFIFO slots simultaneously in this scenario.

Thus, TFIFO slots for ports 0 and 2 get filled faster as compared to Figure A.3, decreasing the chance of underflow at these ports. This agrees with our evaluation results; the egress task does not cause port underflows with packet transmission on only two of the four ports, and can handle close to line rate on two ports. In the new design, we take advantage of this observation by running the egress task on two microengines to handle line rate transmission on all four ports. We statically divide the TFIFO slots, the transmit queues, and the transmit ports between the two Egress microengines to avoid synchronization overheads between them. The next section describes this design in detail.

## A.3 The Two-microengine Egress Design

In this section, we describe the design of the egress task that uses two microengines. In the previous section, we saw that since the three fill threads cooperate to move packets from memory to TFIFO slots for all four ports, they could fill the TFIFO slots at a faster rate when packets were transmitted across only two of the four ports. This was because the fill threads which were unused for ports without packets, were used to fill up slots for the ports which had packets to transmit. Our evaluation for LinkEM shows that the egress task causes port underflows when transmitting on four ports, but can handle line rate transmission on two 100-Mbps ports.

Thus, three fill threads are enough to handle line rate transmission on two ports. This provides the motivation for using two microengines for the Egress task: three fill threads on one microengine can handle transmission on two of the four ports, while the three fill threads on the other microengine can handle the remaining two ports. Also, if we split the ports, the TFIFO slots, and the transmit queues such that there is no synchronization needed between the two microengines, then we can expect near line rate transmission on all four ports using this design.

Figure A.5 shows the TFIFO/port/fillthread mapping for this design. We keep the scheduler-fill thread model on both microengines (MEs). Thus one thread on each microengine runs the scheduler code, while the other three threads on each microengine run the fill thread code. To reduce synchronization overhead between the two microengines, we statically map the transmission of packets on ports 0 and 2 to Egress ME 1, and transmission of packets on ports 1 and 3 to Egress ME 2. Thus the scheduler thread on ME 1 runs round-robin algorithm over queues for ports 0 and 2, while the fill threads on the microengine move packets from memory to TFIFO slots allocated for ports 0 and 2 (all even-numbered slots). Similarly, the scheduler thread on ME 2 runs round-robin algorithm over queues for ports 1 and 3, while the fill threads move packets from memory to TFIFO slots allocated

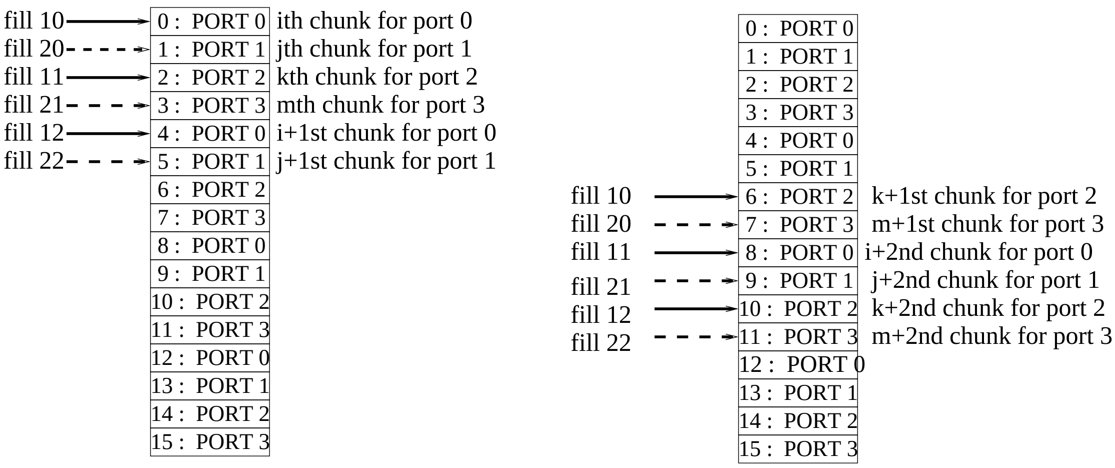


**Figure A.5**. TFIFO/port/fill thread mapping with two microengines for egress for ports 1 and 3 (all odd-numbered slots). In the figure, "fill uv" represents fill thread v running on microengine u. Fill threads on both MEs jump by 6 slots to go to their next assigned slot. As seen in the figure, the $i^{th}$ and $i+1^{st}$ chunks for a packet destined for port 0 are filled at the same time; this is true for packets destined to other ports too (similar to packet transmission on only two ports in the one-microengine Egress design, see section A.2). Thus with this design, the egress task should be able to transmit at near line rate for all four 100-Mbps ports.

# REFERENCES


[1] J. S. Ahn, P. B. Danzig, Z. Liu, and L. Yan. Evaluation of TCP Vegas: Emulation and Experiment. In *Proc. of SIGCOMM '95*, pages 185–195, Cambridge, MA, Aug. 1995.

[2] M. Allman, A. Caldwell, and S. Ostermann. ONE: The Ohio Network Emulator. Technical Report TR-19972, School of Electrical Engineering and Computer Science, Ohio University, Aug. 1997. http://ctd.grc.nasa.gov/-5610/networkemulation.html.

[3] K. Baclawski. NET: A Network Emulation Tool. In *Symposium of Simulation of Computer Networks*, 1987.

[4] L. S. Brakmo, S. O'Malley, and L. L. Peterson. TCP Vegas: New Techniques for Congestion Detection and Avoidance. In *Proc. of SIGCOMM '94*, pages 24–35, London, UK, May 1994.

[5] A. Campbell, S. Chou, M. Kounavis, V. Stachtos, and J. Vincente. NetBind: A Binding Tool for Constructing Datapaths in Network Processor-Based Routers. In *Proc. of the Fifth IEEE Conf. on Open Architectures and Network Programming*, New York, June 2002.

[6] D. Comer. *Network Systems Design Using Network Processors*. Prentice Hall.

[7] P. Crowley, M. Fluczynski, J.-L. Baer, and B. Bershad. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proc. of the International Conf. on Supercomputing*, 2000.

[8] ENP-2505 IXP1200 Development Board. http://www.radisys.com/-oem_products/ds-page.cfm?productdatasheetsid=1055.

[9] K. Fall. Network Emulation in the VINT/NS Simulator. In *Proc. of the 4th IEEE Symposium on Computers and Communications*, 1999.

[10] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993.

[11] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Aubstrate for OS and Language Research. In *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, October 1997.

[12] S. Huang, R. Sharma, and S. Keshav. The ENTRAPID Protocol Development Environment. In *Proc. IEEE INFOCOM 1999*, New York, Mar. 1999.

[13] D. B. Ingham and G. D. Parrington. Delayline: A Wide-Area Network Emulation Tool. *Computing Systems*, 7(3):313–332, 1994.

[14] Intel Corporation: Introduction to the Auto-Partioning Programming Model. ftp://download.intel.com/design/network/papers/25411401.pdf.

[15] IXP1200. http://developer.intel.com/design/network/products/npfamily/-ixp1200.htm.

[16] Intel Corporation: Intel IXA SDK ACE Programming Framework Developer's Guide, June 2001.

[17] IXP2400. http://developer.intel.com/design/network/products/npfamily/-ixp2400.htm.

[18] IXP2800. http://developer.intel.com/design/network/products/npfamily/-ixp2800.htm.

[19] V. Jacobson. Berkeley TCP Evolution from 4.3-Tahoe to 4.3-Reno. In *Proc. of the Eighteenth IETF*, Vancouver, 1990.

[20] E. Johnson and A. Kunze. *IXP2400/2800 Programming*. Intel Press.

[21] P.-H. Kamp and R. N. M. Watson. Jails: Confining the Omnipotent Root. In *Proc. 2nd Intl. SANE Conference*, May 2000.

[22] S. Keshav. Packet-Pair Flow Control. Technical report, Murray Hill, New Jersey, 1994.

[23] K. Mackenzie, W. Shi, A. McDonald, and I. Ganev. An Intel IXP1200-Based Network Interface. In *Workshop on Novel Uses of System Area Networks at HPCA*, 2003.

[24] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. Research Report 95/8, Digital Equipment Corporation Western Research Laboratory, Dec. 1995.

[25] NIST Internetworking Technology Group. NIST Net Home Page. http://-www.antd.nist.gov/itg/nistnet/.

[26] NLANR Distributed Applications Support Team. IPERF Home Page. http://dast.nlanr.net/Projects/Iperf/.

[27] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proc. of SIGCOMM*, Sept. 1997.

[28] V. Paxson. On Calibrating Measurements of Packet Transit Times. In *Measurement and Modeling of Computer Systems*, pages 11–21, 1998.

[29] V. Paxson. End-to-End Internet Packet Dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, 1999.

[30] L. Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols, 1997.

[31] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. Canberra, Australia, June 2000.

[32] T. Spalink, S. Karlin, L. Peterson, and Y. Gottlieb. Building a Robust Software-Based Router Using Network Processors. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 216–229, Chateau Lake Louise, Banff, Alberta, Canada, October 2001.

[33] The VINT Project. *The* ns *Manual*, Apr. 2002. http://www.isi.edu/nsnam/-ns/ns-documentation.html.

[34] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 271–284, Boston, MA, Dec. 2002. USENIX ASSOC.

[35] S. Wang and H. Kung. A Simple Methodology for Constructing an Extensible and High-Fidelity TCP/IP Simulator. In *Proc. IEEE INFOCOM 1999*, New York, Mar. 1999.

[36] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, Dec. 2002.

[37] I. Yeom. ENDE: An End-to-End Network Delay Emulator. Master's thesis, Texas A&M University, College Station TX, 1998.