EXPLOITING NETWORK PROCESSORS FOR LOW LATENCY, HIGH THROUGHPUT, RATE-BASED SENSOR UPDATE DELIVERY

By

Kim Christian Swenson

A thesis submitted in partial fulfillment of
the requirements for the degree of

Master of Science in Computer Science

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

DECEMBER 2009

To the Faculty of Washington State University:

The members of the Committee appointed to examine the thesis of KIM CHRISTIAN SWENSON find it satisfactory and recommend that it be accepted.

David E. Bakken, Ph.D., Chair

Carl Hauser, Ph.D.

Min Sik Kim, Ph.D.

ACKNOWLEDGMENT

Thanks to my committee members and in particular Dr. David E. Bakken and Dr. Carl Hauser for all their time, patience, advice and direction. Thanks to Sidra Gleason with the EECS Graduate Student Services for her invaluable assistance. Thanks to The School of Electrical Engineering and Computer Science as well as the Graduate School of Washington State University for allowing me to pursue a master's degree. Thanks to The Norwegian State Loan Fund for substantial financial support throughout my studies. Thanks to my employer TOMRA for allowing me a temporary reduction in work time so I could complete this thesis. Thanks to Intel Corporation and in particular Dr. David Tennenhouse for donating network processor hardware. Thanks to Netronome for providing me with an updated version of the Intel Internet Exchange Architecture Software Development Kit. Thanks to all my good friends for their encouragement. Last but not least, thanks to my family for their unconditional love and support.

EXPLOITING NETWORK PROCESSORS FOR LOW LATENCY,

HIGH THROUGHPUT, RATE-BASED SENSOR UPDATE DELIVERY

Abstract

by Kim Christian Swenson, M.S.
Washington State University
December 2009

Chair: David E. Bakken

GridStat is a novel publish-subscribe middleware framework which delivers rate-based streams of sensor updates for the power grid. Its novelty includes providing different subscribers to the same sensor update stream with different guarantees in terms of end-to-end delay, rate, and number of disjoint delivery paths. GridStat implementations on a general-purpose, dual-core CPU have very good per-hop forwarding latency on the order of 0.1 milliseconds. However, they can only support approximately 20 thousand forward/sec. In this thesis, we consider the problem of how to exploit high-speed network processors to increase the forwarding throughput for a GridStat status router (publish-subscribe forwarding engine). This thesis presents an analysis of the problem, and a design for GridStats routing algorithm on top of the Intel IXP 2400 series network processor. Experimental analysis shows how it is possible to support up to approximately 1 million forwards/second with a forwarding latency of approximately 0.01 milliseconds.

Table of Contents

1 Introduction **1**

2 Background **3**

2.1 GridStat 3

2.1.1 Routing algorithm pseudo code 6

2.2 Hardware platform 7

2.2.1 Network Processors in general 7

2.2.2 The Intel IXP2400 architecture 7

2.3 Related work 15

2.3.1 GridStat 15

2.3.2 Network processors 16

2.3.3 Multicast IP routing 17

3 Prototype design and implementation **19**

3.1 Implementation Challenges 19

3.1.1 Router tailored hardware 20

3.1.2 Parallelism and throughput 22

3.1.3 Predictable packet delay 24

3.1.4 Scalability of throughput 24

3.2 Mapping the GridStat algorithm on network processor hardware 25

3.3 Microblocks 30

3.3.1 General packet flow 30

3.3.2 Packet Receive (RX) microblock 31

3.3.3 Packet Transmit (TX) microblock 32

3.3.4 `Route` microblock 34

3.4 IXP2xxx synchronization and locking mechanisms 44

3.5 Constraints on the IXP2xxx hardware 45

3.5.1 Soft constraints 45

3.5.2 Hard constraints 46

3.6 Features not included in the status router prototype 47

4 Analysis and Simulations **49**

4.1 Experiment configuration details 50

4.2 Performance measurements 53

4.3 Experiments overview and expectations 55

4.4 Experiment 1 60

4.4.1 Setup 60

4.4.2 Analysis 60

4.5 Experiment 2 61

4.5.1 Setup 61

4.5.2 Analysis 64

4.6 Experiment 3 64

4.6.1 Setup 64

4.6.2 Analysis 69

4.7 Experiment 4 73

4.7.1 Setup 73

4.7.2 Analysis 73

4.8 Experiment 5 73

4.8.1 Setup 73

4.8.2 Analysis 76

4.9 Summary 76

4.9.1 Throughput 76

4.9.2 Latency 81

4.9.3 Jitter 81

4.9.4 Algorithm throughput scalability 82

4.9.5 Conclusions 82

5 Conclusions **85**

5.1 Performance and constraints 86

5.2 Future work 91

References **91**

A Code Listing **95**

A.1 PacketStreamGenerator.java 95

A.2 RTSetupCodeGenerator.java 98

A.3 Packet.java 104

A.4 GraphProducer.java 107

A.5 IXPSimulationLogAnalyser.java 116

A.6 utilities.uc 133

A.7 routing_table.uc 136

A.8 packet_echo.uc 143

List of Figures

2.1 GridStat middleware overview 4

2.2 Schematic overview of the IXP24xx network processor 8

2.3 General packet flow 15

3.1 Prototype packet flow 29

3.2 Routing table data structure 38

3.3 Routing table entry details 39

3.4 Original subscriber list 39

3.5 Collapsed subscriber list 40

3.6 CAM structure 41

4.1 $IdealThroughputCapacity(x)$ 59

4.2 Simulations for experiment 1 62

4.3 Slopes of experiment 1 63

4.4 Simulations for experiment 2 part 1 65

4.5 Simulations for experiment 2 part 2 66

4.6 Slopes of experiment 2 67
4.7 Simulations for experiment 3 part 1 70
4.8 Simulations for experiment 3 part 2 71
4.9 Slopes of experiment 3 72
4.10 Simulations for experiment 4 74
4.11 Slopes of experiment 4 75
4.12 Simulations for experiment 5 77
4.13 Slopes of experiment 5 78
4.14 Throughput capacity as a function of inner loop iterations 80
4.15 Comparing status router performance 84

List of Tables

4.1 Mapping e to $RuntimeComplexity(RoutingTable(e))$ 57

4.2 Incoming and outgoing inter-packet-delay per simulation of experiment 1 . . 64

4.3 Incoming and outgoing inter-packet-delay per simulation of experiment 2 . . 68

4.4 Incoming and outgoing inter-packet-delay per simulation of experiment 3 . . 69

4.5 Incoming and outgoing inter-packet-delay per simulation of experiment 4 . . 76

4.6 Incoming and outgoing inter-packet-delay per simulation of experiment 5 . . 79

Chapter 1

Introduction

Third generation network processors are high performance network solutions designed for efficient packet processing and deep packet inspection. A third generation network processor generally include a physical network interface with multiple links, several RISC processors to process packets along a fast data path, and an embedded general processor to perform higher level protocol management [4]. Network processors can take on different networking roles, including real-time encryption, protocol translation, or multicast routing. Network processors are a relatively new technology aimed to combine the high performance of expensive but less flexible ASIC solutions with the low cost and flexibility of software based packet processing. A reduced instruction set tailored for packet processing and scalability through parallelism and pipelining are the main design ideas that enable the high performance of a network processor. Although the most demanding backbone networks still require custom hardware, network processors deliver high enough performance to meet many networking demands and by far outperforms any regular computer based packet processing system.

GridStat is a status dissemination middleware framework designed to manage the flow of power grid status data in real-time [15, 10, 22, 6, 2, 7, 5]. GridStat use publish and subscribe concepts to hide the details of an advanced management, rate-filtering and routing middleware system. GridStat is designed ground-up to support the real-time delivery of very high bandwidth data streams, achieved among other things through early rate-filtering and efficient multicasting. GridStat supports quality of service (QoS) solutions like reliable timely delivery and secure communication.

In this thesis we explore some of the performance potential and scalability challenges of implementing a GridStat status router using network processors. Of particular interest is how the current GridStat rate-filtering and routing algorithm [15] scales in a network processor as throughput and routing-table complexity increases. The design and implementation of a prototype of a GridStat status router using the Intel Internet Exchange Architecture (IXA) is explained in detail in chapter 3. Chapter 4 present our findings from simulation and analysis, and chapter 5 suggests some subscription constraints that would enable GridStat status routers to provide latency and throughput guarantees.

Chapter 2

Background

2.1 GridStat

GridStat is a flexible publish-subscribe middleware designed for real-time delivery of power grid status information [19, 14, 10, 22, 6, 7]. Users of the middleware need not concern themselves with the details of the data delivery plane, rather they need only relate to either writing updates to variables (publisher role) or receiving updates of selected variables (subscriber role). Figure 2.1 show publishers writing updates to variables X, Y, and Z, and subscribers receiving updates of selected subsets of these variables. Each node in the directed graph of the data delivery plane is called a status router, it's the performance of such a GridStat status router implemented on network processors that is the focus of this thesis.

GridStat has more than one mode of operation, allowing the middleware to gracefully degrade service in the event of congested or damaged communication channels. Subscribers of

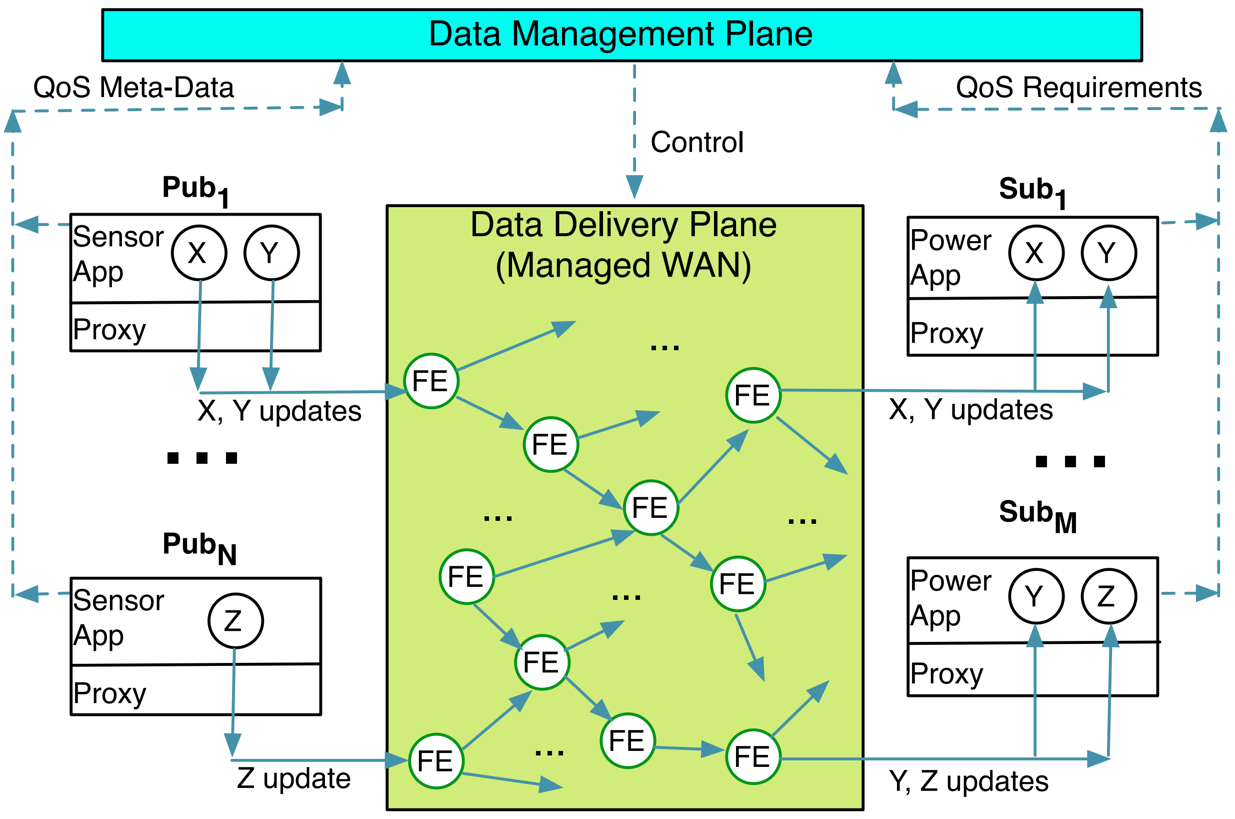

Figure 2.1: GridStat middleware overview

GridStat still receive useful and correct status information when the network runs at a lower level of operation, though likely receiving updates less often. A GridStat status router is a leaf node in the GridStat management hierarchy where all status routers make an interconnected graph to compose the possible data paths of power grid status data. These status routers all run the same algorithm for rate filtering and multicast routing, and differ only by where in the management topology they reside, therefore using different routing table information.

2.1.1 Routing algorithm pseudo code

```
 1: loop
 2:     $P \leftarrow GetNextPacket()$
 3:     $V \leftarrow StatusVariableID(P)$
 4:     $Pub \leftarrow PublishInterval(V)$
 5:     for $L$ in $RouterLinks()$ where $L \neq IncomingLink(P)$ do
 6:         $S \leftarrow SubscriberList(V, L)$
 7:         for $s$ in $S$ do
 8:             $R \leftarrow (Timestamp(P) + Pub/2)$ $modulo$ $SubscriberInterval(s)$
 9:             if $R < Pub$ then
10:                 Forward packet $P$ on link $L$
11:                 continue for loop at line 5
12:             end if
13:         end for
14:     end for
15: end loop
```

Throughout this document lines 5 to 14 are referred to as the algorithm's outer loop and lines 7 to 13 as the algorithm's inner loop. The subscriber list at line 6 is ordered ascending on subscriber interval values.

2.2 Hardware platform

2.2.1 Network Processors in general

In principle, a network processors is a network capable device that specializes in processing packets. Network processors are generally categorized in three generations [4]. The first generation of network processors are packet processing systems built with conventional computer systems. The second generation of network processors added specialized processors more directly connected to the physical network interface itself allowing a fast data path and much higher response times and throughput capacity. Third generation network processors extend the second generation by also including an embedded general processor to manage higher level of protocols and to interact with the other processors along the fast data path through shared memory.

2.2.2 The Intel IXP2400 architecture

Details on the IXP2400 network processor can be found in [1].

Intel XScale Core

The Intel XScale core is a 32-bit general purpose processor. The core is compatible with the ARM Version 5 (V5) Architecture with the exception that the XScale core does not provide hardware support of floating point instructions. Features of the XScale include Memory Management (MMU), Instruction Cache, Data Cache, and Interrupt Controller. The core

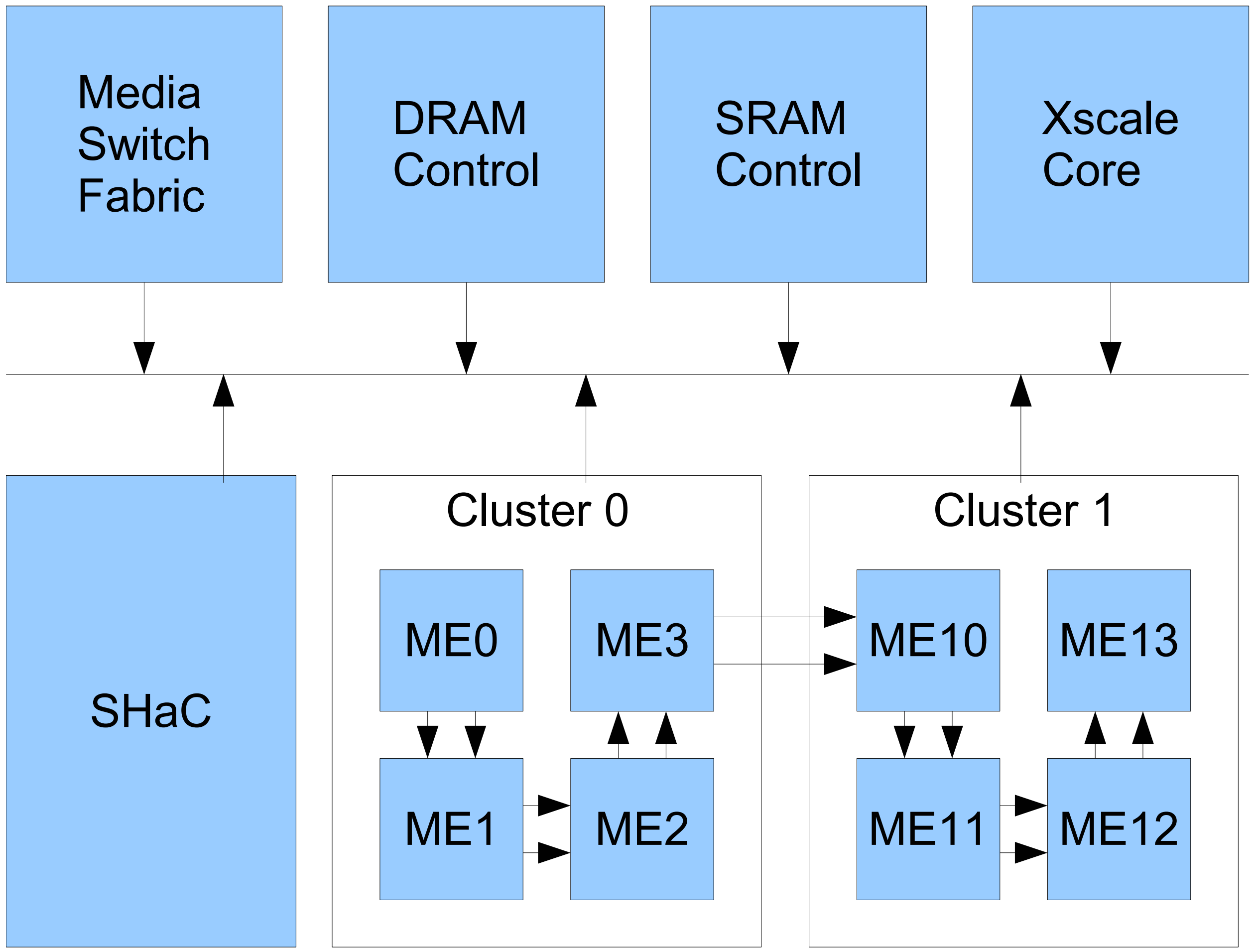

Figure 2.2: Schematic overview of the IXP24xx network processor

has 4 GB of addressable memory, where the 2 GB low addresses map onto DRAM and the 2 GB high addresses map onto other targets like SRAM, PCI, and CSR's. Typical operating system choices are VxWorks® and Linux®.

Microengines

Microengines allow fast programmable packet processing. The IXP2400 has 8 microengines. Each microengine is its own processor with a range of registers available. Key to the speed and flexibility of the Microengines is their fast access to most special purpose subsystems of the IXP2400 chip through buses.

Control Store Microengines in the IXP2400 each has a static RAM control store. It can hold 4096 instructions each 40-bits wide. Before any execution can be done, code must be loaded into this control store. The code store is typically not changed after execution starts, even though it's possible, however patching of symbols is a common operation after code is loaded but before execution starts and allows more dynamic software architectures.

Contexts Each Microengine has 8 hardware contexts available. Each context has its own set of registers, a program counter, and context specific local registers. By default registers are allocated in a per context exclusive manner giving each context their fair share. Naturally this limits the amount of registers available per context but it allows very fast context switches which is key to achieve good performance. A context can only be in 1 of 4 states:

- Inactive. Applications may not require all 8 contexts, when that is the case, select contexts are marked as inactive and will not participate in the context switch rotation. The corresponding available registers may be shared among the remaining contexts.

- Executing. At most one context may be in the executing state at any time. When in the executing state, that context's program counter is used to fetch instructions from the control store. A context will stay in the executing state until it executes an instruction that tells it to sleep.

- Ready. This state indicates that the context is ready to execute. The only reason it cannot execute is because another context is already in the execute state. The next context to execute is selected round robin among the Ready state contexts once the executing context goes to sleep.

- Sleep. Waiting on external event(s). Once all events the context is waiting on has occurred the context is put in the Ready state.

Registers The Microengines of the IXP2400 has several types of registers:

- Local Control and Status Registers (CSRs). These registers hold special purpose status information and is not part of the direct data path registers. There is a delay of 3-4 cycles to read/write Local CSRs. These registers reside in the SHaC unit.

- 256 General purpose registers organized in 2 banks A and B. Most instructions require operands from separate banks, the allocation of registers to banks in software is done by the compiler and not by the programmer.

- 512 Transfer registers. There are 4 types of transfer registers: `S_Transfer_In`, `S_Transfer_Out`, `D_Transfer_In`, `D_Transfer_Out`. These registers are used when accessing data external to the microengine like SRAM, DRAM, and the SHaC.
- 128 Next Neighbor (NN) Registers. Microengines have dedicated data paths (NN registers) between them. These data paths are predefined in which Microengine can send/read data to/from another Microengine.
- 640 32-bit words of Local Memory (LM). Microengine LM is accessed through context copies of the `LM_ADDR` registers. Each context have 2 such registers: `LM_ADDR_0` and `LM_ADDR_1`.

SRAM Controller

There are 2 independent SRAM controllers, each supporting pipelined QDR synchronous static RAM (SRAM). SRAM is ideal for representing packet queues and meta-data and the addresses here are typically mapped directly to packet data in DRAM through some simple arithmetic function. The SRAM is logically 32-bit wide, so special care must be taken by the programmer when access outside this alignment occur.

DRAM Controller

The DRAM controller support a single 64-bit channel of DRAM. 2 GB of address space is allocated to DRAM. The DRAM controller may receive read and write requests from the Intel XScale core, Microengines, and the PCI bus master. These connect to the DRAM controller via the Command Bus and the Push and Pull buses. DRAM on the IXP2400 also support Error Correcting Code (ECC). The DRAM is logically 64-bit wide, so special care must be taken by the programmer when access outside this alignment occur.

Media and Switch Fabric Interface (MSF)

The MSF connect the IXP2400 to a physical layer device (PHY) and/or to a switch fabric. The MSF acts as the bus for SRAM push/pull, DRAM push/pull, and the physical network hardware layer. The MSF allows transparent and uniform access to different king of physical network layers without the need for application code changes except some configuration options.

PCI Controller

The IXP2400 PCI Controller may be used to communicate between the Intel XScale core and its host (any computer with a compatible PCI interface).

SHaC unit

The SHaC unit consist of Scratchpad, Hash Unit, and CAP. These three units all share the Command, S_Push, S_Pull bus interfaces and only one command to either unit can be processed per cycle.

Scratchpad Memory Scratchpad Memory has 16KB general purpose storage. The memory is organized as 4K 32-bit words accessible by the Intel XScale core and Microengines.

Scratchpad is not byte-writable, all read and write operations on Scratchpad must work with 4 byte aligned words of 4 bytes each. Max 16 32-bit words (total of 64 bytes) can be accessed in one read/write operation. Scratchpad support 16 hardware assisted rings. These rings are fixed size FIFO structures using head and tail pointers and the hardware operations on them are atomic. The size of the rings must be decided at compile or load time and must be one of 128, 256, 512, or 1024 32-bit words. Scratchpad allows much faster memory access than SRAM or DRAM, and is therefore ideal for small data structures accessed frequently.

Hash Unit The Hash Unit take 48, 64, and 128-bit data and produce a 48, 64, and 128-bit hash index respectively. The Hash Unit can be accessed by Microengines and the Intel XScale core. The hash unit is ideal when doing searches with large keys like L2 addresses.

Control and Status Register Access Proxy (CAP) The CAP has many control and status registers (CSRs) configuring network processor behavior. In addition to configuring the hardware, the CSRs provide mechanisms for inter-thread communication across microengines and ways to communicate between the XScale core and microengines. Note that Scratchpad rings and SRAM memory can also be used to communicate between the XScale core and the microengines.

Performance monitor

Intel XScale core peripherals (XPI)

The following peripherals fall under this category: Interrupt Controller, Timers, GPIO, UART, and SlowPort.

Interrupt Controller The Interrupt Controller provide ability to enable or mask interrupts from sources like timers, microengine software, PCI device, hardware error conditions and more. Reading interrupt status and enabling/masking interrupts are done as read and write operations on memory mapped registers.

Timers 4 programmable 32-bit timers are provided. They can be clocked by the internal clock, a divided version of the clock, or by a signal on an external GPIO. Periodic timers can generate interrupts if programmed so.

GPIO The IXP2400 has 8 General Purpose IO (GPIO) pins. Usage of GPIO pins range from output LED indicators or input switches. GPIO pins can provide interrupts to the Intel XScale core or clocking the timers.

UART There is a RS-232 compatible Universal Asynchronous Receiver/Transmitter (UART) usable by a maintenance console or a debugger.

SlowPort The SlowPort allows access to the Flash ROM and asynchronous external devices at 8, 16, or 32-bit access rate.

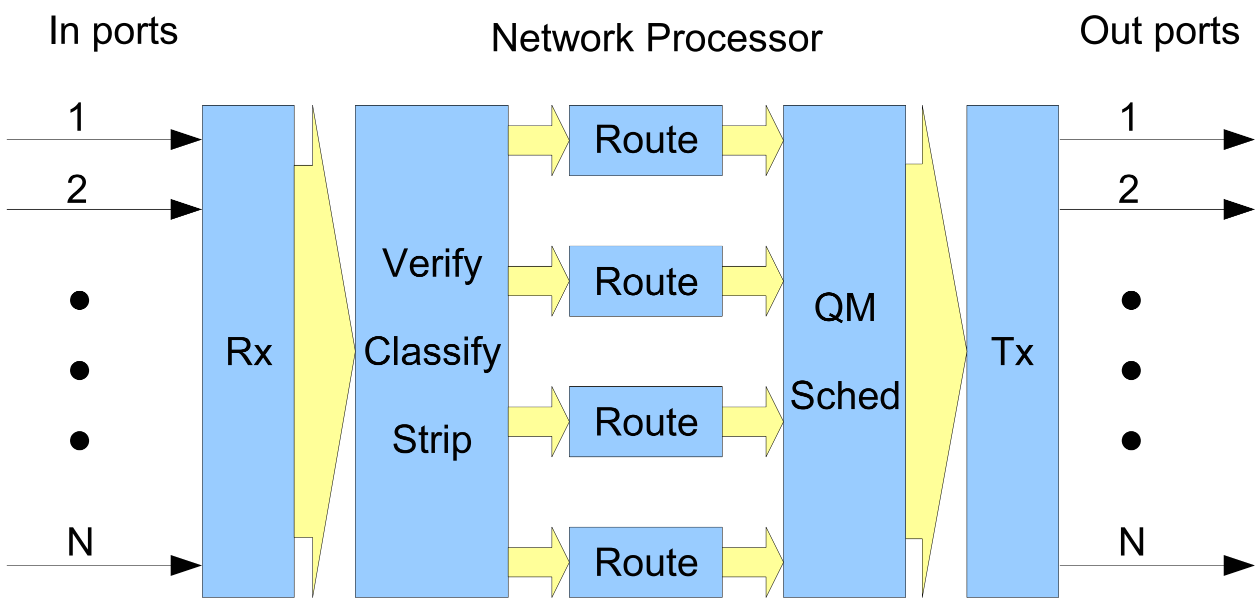

Figure 2.3: General packet flow

2.3 Related work

2.3.1 GridStat

An initial prototype of the GridStat framework, including a status router written in Java, was developed in 2000 [5, 13]. Since then, the GridStat framework has seen many enhancements and extensions, particularly related to security, quality of service (QoS) and ease of use. In 2005, it was shown that GridStat running on a regular computer with Java based status routers could satisfy the industry end-to-end latency requirement for delivering phasor measurement unit data through the GridStat middleware at relatively low throughput rates [14]. In 2008, a C implementation of a GridStat status router proved capable of more than twice the throughput than current Java implementations and with lower and far more predictable delays [19]. While computer based status routers seem to be capable of thousands and even tens of thousands of forwards per second, network processors (of comparable vintage) are expected to be capable of many million forwards per second and at much lower latencies and better predictability than regular computer based implementations.

2.3.2 Network processors

Third generation network processors have been widely available to industry and academia for almost ten years now, and a lot of interesting research has been done with them.

The IXP2400 network processor has been used for studies on various topics. Bursty traffic and a multi threaded caching algorithm [3]. Packet reordering algorithms that are more efficient than those that come bundled with the SDK [8]. Automated application design and mapping of code onto different processing elements [20]. Network processor runtime operating system to dynamically reconfigure and switch code running in the data path of the network processor based on traffic patterns [25].

The IXP2800 and IXP2850 are other very popular network processors for studying specific applications or general methods of network processor architecture. Common applications to study are packet and flow classification algorithms [16, 27, 21] which must eventually be part of a GridStat status router if it is to handle unexpected input properly. Such classification algorithms were also studied along with fast TCP session creation and packet reordering. A fast IPv6 forwarding and routing algorithm have been demonstrated [11], while at the same time analyzing some general IXP network processor architecture programming challenges and how to best maintain packet ordering and hide internal IO latency. There are also other studies on hiding the internal latency of memory and other subsystems through multi-threading techniques [9].

Parallelization and multi-threading is key to performance and scalability of network processors in general [26]. There exists methods and applications to compare different network processor platforms, which was tested to work on the IXP1200 [18].

There is also much research done on automatic and dynamic mapping of network processor code onto microengine as runtime requirements change. Methods to do this has been developed through both pure analysis [24] and statistical analysis [17]. There even exists methods for comparing different dynamic runtime systems with each other [12].

2.3.3 Multicast IP routing

Multicast IP is a facility that send an IP packet to a group of hosts identified by a group address [23]. Multicast routers are a subset of IP routers that manage and provide the multicast IP facility. In a router with N physical network ports of equal capacity, the outgoing throughput is potentially up to $N - 1$ times bigger than the incoming throughput. This is because that the most expensive case of multicast is a broadcast, where each incoming packet is sent to every outgoing port expect through the port it arrived. In the case of sustained data streams where all packets are broadcasted, the router will not be able to accept an incoming throughput of more than the capacity of a single port. A simplified explanation proving this is as follows: Observe that when a router receives incoming data streams at rates higher than a single port, sending every such incoming packet to every other port would exceed any other port's capacity. The real multicast world is generally more complex than such an ideal simplification of the worst case. It may be expected that a real-world multicast traffic is bursty and would not necessarily trigger a full broadcast in any router. Such environments can be very complex and the throughput capacity and packet processing time are dynamic and can only be described through advanced network traffic and routing buffering analysis and would most likely also require simulation or real-world observation to be well described.

Chapter 3

Prototype design and implementation

3.1 Implementation Challenges

Implementing the GridStat Status Router on an Intel IXP network processor had some challenges:

- Complexity of such a highly parallel programming model as the one presented through the IXA SDK.
- In fixed rate streams, predictable packet delay is important.
- Exposure to network processor hardware complexity, e.g. many layers of memory but no MMU, some special instructions share resources across microengines.
- No hardware support for division or modulo operations used by the routing algorithm.

3.1.1 Router tailored hardware

The Intel IXP network processors are designed with routing applications in mind.

The IXP network processors have a dedicated hardware/software layer called fast path or data-plane. The hardware at this layer is very fast and designed with fast packet processing and forwarding in mind. The efficient way packet data is transferred from physical links to DRAM and sent out again is one of the best examples of dedicated hardware for packet processing. Hardware supported atomic ring buffer operations on SRAM, fast atomic synchronization primitives in scratch memory, and next-neighbor registers are all examples of hardware support for a fast and reliable software pipeline at the data-plane.

Perhaps the most obvious benefit of using a network processor chip like the IXP family is the presence of many processors running in parallel each with multiple threads making it possible to hide the waiting on internal hardware IO. Some of the benefits and drawbacks of the parallel nature of the IXP network processor are discussed in section 3.1.2.

Typical to any routing application is the presence of a semi-static routing table data structure. Navigating the routing table data structure has to be fast at the data-plane where table lookup is performed. Hashing has proven a fast method to navigate large data structures and the IXP2xxx chips provide hardware accelerated hashing using either 48-bit, 64-bit, and 128-bit keys and hashes. Hashing is the method of choice in navigating the routing tables of all the sample routing applications including IP and multicast IP. Hashing remain the obvious choice for navigating the routing table in a GridStat status router, and the one-to-one relationship between variable-ID (32-bit flow membership identifier) and the corresponding complete routing table entry makes mapping a hashing algorithm onto navigating the routing table in GridStat straightforward.

The IXP2xxx chip provide a small but fast mechanism to support a microengine-local cache available at all microengines. By combining the 640 4-byte words of microengine local memory and the 16 entry Content Addressable Memory (CAM) it's easy to store 16 entries of a large data structure (like a routing table) in local memory and using the fast CAM to navigate it. Although the cache often prove an effective performance enhancement to an application like IP routing and forwarding, it's not given that it will in the GridStat application. The benefit of a cache is much greater in an application with a somewhat unpredictable and bursty traffic environment. General Internet IP traffic has this exact nature, due to it being used by such a diverse number of applications often with unpredictable users dictating traffic and bursts. The GridStat application however, is neither bursty nor unpredictable in its steady state. The publish/subscribe architecture is designed with reliable traffic streams that are not bursty, rather they are fixed rate streams. Only when alerts in GridStat are issued will we see bursty behavior, alerts are not discussed in this thesis. Given a reasonably large amount of publish streams where many of them operate with the same or similar publish rates, we can not expect a cache to give much benefit. On the contrary, In GridStat, we might expect the routing table cache to be a lookup overhead for most publish streams. It should be noted that in the case of a few (less than some fraction of the cache size) very fast (high rate) publish streams, these fast streams might always get a cache-hit while most other streams get a cache-miss, hence a cache will favor the fast streams. This caching problem in

GridStat could have other and better solutions, one of which would be to dedicate certain microblocks to some high rate streams giving it the benefit of always having the relevant routing table entries in local memory.

3.1.2 Parallelism and throughput

Parallelism in a network processor when applied to an application like the GridStat status router, is always about processing more than one packet at a time and at the same section of the packet processing pipeline. Given a simple packet processing pipeline consisting of `Receive`, `Route`, `Queue Manager`, and `Transmit` microblocks. The fact that these microblocks run in parallel give no speedup benefit when we look at the processing delay of a single packet. Rather, the packet processing pipeline allows high throughput as we can have many packets in the pipeline each being processed by a different part at the same time. It's important to note that having one processor doing all steps on every packet would yield the same single packet delay as that of a parallel pipeline, however the throughput when using multiple processors each with a separate task is much higher. Parallelism can also be extended within part of the pipeline by having one of the microblocks (steps) use more than one microengine (processor). An example would be having the `Route` microblock use four microengines. Each microengine would run the same code but always process different packets allowing more packets to pass through this step of the packet processing pipeline over the same time interval. Any given packet would only visit one of the four instances of the `Route` microblock.

Ideally, the ability to process packets in parallel will give a speedup corresponding with the degree of parallelism. e.g. Running the `Route` microblock on three microengines simultaneously allow three times the amount of packets to be routed over the same time interval than running `Route` on one microengine only. If we assume that the task of dividing and combining work takes no time and that synchronization between the microengines running the same microblock is not needed, we get a throughput scalability where adding one microengine give one speedup. In GridStat, the routing tables might change at runtime and packet order must be maintained, hence processing many packets in parallel require some synchronization to access the routing table and to maintain packet order.

There are two levels of parallelism within a microblock running on multiple microengines. The most obvious one is the fact that when you have more than one microengine, these can execute instructions at the same physical time providing true parallelism. The other not so obvious one comes from the fact that each microengine have eight hardware supported threads (contexts) each with its own program counter and fair share of the registers. As the threads share the same physical processor they can't execute instructions at the same physical time, however some part of packet processing don't require executing instructions. Waiting on IO is something that most processing code must do, and while waiting on IO, a thread will yield, allowing other threads to execute while it's waiting. This is how "hiding" IO delay is achieved, and although any "waiting on IO" will add to the delay of the corresponding packet being processed, the delay does not "steal" throughput capability of the application as long as the total amount of "waiting on IO" delay in a microengine per packet is no more than eight times (as there are eight threads) that of the delay added by executing instructions for that packet in the microengine.

3.1.3 Predictable packet delay

Data passed through the GridStat framework is of a “time-critical” nature as some users / systems expect to act quickly based on what the data describes. Ideally all packets would have zero delay allowing systems to respond immediately. Clearly we can’t obtain zero delay in such a framework so the question becomes “how well can we do.” The fixed-rate nature of traffic streams intended for the GridStat framework would also suggest that we want predictable delays as the streams will only stay as “fixed-rate” as the variance in delay from packet to packet is. As the users of GridStat will need to obtain “global-snapshots” using data from multiple streams, the snapshots can only be viewed as soon as all the packets belonging within that certain time-window defined as a single snapshot have reached the subscriber.

3.1.4 Scalability of throughput

Scalability within the status router itself depend on the routing algorithm used. The GridStat status router routing and rate-filtering algorithm (see section 2.1.1) is simply referred to as “algorithm” or “routing algorithm”. Each published variable in the GridStat architecture is called a “flow” and each flow has an associated unique “flow identifier.” Every flow has a routing table entry which contains a two-dimensional list of links and values indicating subscriber interval periods. For a given flow, each link has a set of 0 or more subscriber values. This set we call the "collapsed subscriber interval list," see 3.3.4 for a more detailed explanation of this set.

The original routing algorithm scale based on the number of links on the status router and on the collapsed subscriber interval list. Changing the number of links on a status router falls into the category of managing the status router hierarchy as a whole. From a single status router's point of view, the number of physical links to use is dictated by the management layer. The original routing algorithm assume that subscribers may register to receive published streams at any rate (using millisecond resolution.) The nature of these constraints will in the worst-case cause packet delay to scale at $O(mn)$ where m is the number of links and n is the number of subscribers. This is because the algorithm will always iterate over all outgoing links and for each link iterate through every item in the associated collapsed subscriber interval list. The constant time required in each iteration is high as well, given that each operation requires a modular (division) operation which is not even supported by IXP2xxx hardware.

3.2 Mapping the GridStat algorithm on network processor hardware

Given the Java version of the GridStat algorithm, mapping this algorithm onto IXP hardware has proven quite the challenge. Many things about network processor hardware forces specific design on aspects of the software. The most obvious are that all the processing units (microengines) of the network processor runs in parallel. Secondly is the exposure of 4 layers of RAM to the programmer, DRAM, SRAM, Scratch, and Microengine local memory. And the last major challenge is the use of a a programming language called “Microcode” which is a kind of macro’ed assembly with some simple block constructs.

Choosing the programming language

There exist a compiler for a higher level programming language called “Micro C” which is very close to standard “C” but with some special added constructs to support the IXP hardware. In the prototype used in this thesis Microcode was chosen since most of the current SDK versions were already written using Microcode. Most of the development tools also seem to support Microcode better than Micro C which also made the decision easier. Examining sample projects written in both languages reveal that the amount of work on the programmers part is somewhat lighter in Micro C than it is in Microcode, but not as significant a difference as usually found between assembly and C-code in general programming. Micro C is easier to read and also hide as much hardware exposure as possible giving it a more abstract feel than Microcode. Ultimately the compatibility with libraries and development tools along with help material available made the choice fall on Microcode. However, in hindsight it’s easy to feel that the implementation would have been easier to do in Micro C, and perhaps that should be the choice of a production implementation in order to favor code clarity over complete control of every microengine instruction.

Working with multiple processors

The IXP2400 platform has 8 microengines per network processor. Each microengine is its own processor running in perfect clock synchronization with all the other microengines. As the packet processing done in GridStat is fairly expensive the code per microblock should utilize more than one microengine to give a significant performance advantage over using only one microengine. This performance advantage only comes to its right when traffic loads are high, at low traffic rates there is no benefit of using more than one microengine per microblock. The implementation used for simulations in this thesis (see chapter 4) utilize two microengines for the routing algorithm, although this is probably not optimal, it suffices and ease the performance analysis a little since a slower moving system is easier to measure.

Designing the GridStat status router

The IXP hardware platform provides the programmer with full freedom as to how software should run. Intel however, in designing the IXP features and structure as they have, did so optimizing its features to support some special ways of designing software. Given all the example software provided in the IXA SDK, it's easily demonstrated that this specific way of designing software works well for most common network applications. We have chosen for the GridStat status router implementation to use the software design principles demonstrated in the IXP/IXA documentation that follows the network processor's software development kit.

The introduction of a "microblock" appear as a chief characteristic of applications on the IXP. The IXA support the concept of microblocks in many ways to streamline implementing them. A microblock is a software component in the packet processing pipeline performing a specific operation on packets. Conceptually the core of each microblock is an infinite loop called a "dispatch loop" like:

```
// dispatch loop start:
(1) Source packet P
(2) Process packet P
(3) Sink packet P
// dispatch loop end.
```

A packet processing pipeline consists of many microblocks chained together. Consider a pipeline with microblock A and microblock B each with a dispatch loop as the one above. In order to chain A and B together we can create a shared data structure AB_RING with FIFO operation logic. Now A will sink packets onto the AB_RING and B will source packets off the AB_RING. The data structure used to link microblock together in this way is usually a ring buffer that reside in scratch memory called a "scratch ring". These rings are an excellent choice as they reside in the fastest shared RAM available big enough to support such structures for large ring buffers. Scratch rings are easily accessed by all microengines using fast atomic hardware instructions to get and put data on them solving all synchronization issues with shared ring structures. As these rings are fixed size they might become full or empty. It's the responsibility of the microblock sinking a packet on a ring to check whether the ring has available space before performing a put operation. If the ring is full, the packet is either dropped or the microblock stall waiting until there is space. In order to support reusing microblocks across applications the specific assignment of scratch rings in between

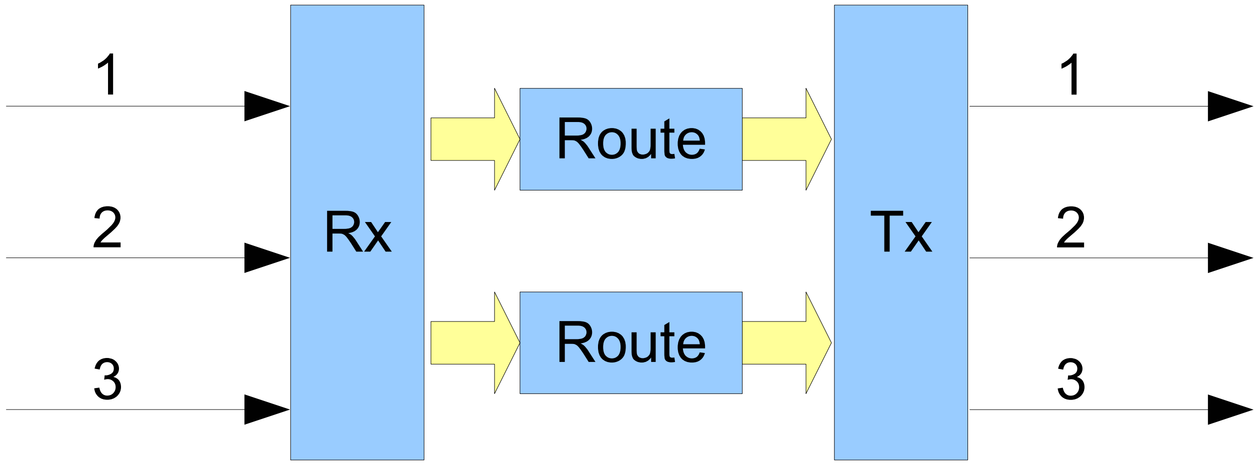

Figure 3.1: Prototype packet flow

microblocks can be set at load time which is after microblock compilation but before the code starts executing. This enables an application designer to reuse pre-compiled microblocks in their own packet pipeline by patching scratch ring assignments into the code after it's loaded into the microengine hardware but before code execution starts.

The GridStat status router application use two microblocks taken from the IXA and one microblock specific to the application. The `RX` and `TX` microblocks taken from the IXA provide endpoints in the packet processing pipeline as they are responsible for receiving and sending packets on the physical medium. The GridStat features are all implemented within the `Route` microblock which provide the core route and filtering logic used in the GridStat routing algorithm. This is a simple design that can be improved on, but suffice to demonstrate some performance properties of using network processors to implement status routers. See figure 3.1 for a prototype architecture overview.

3.3 Microblocks

3.3.1 General packet flow

Packets arrive through physical links and all enter the GridStat software pipeline through the `Ethernet Receive` driver (`RX`) running on a single microengine. `RX` put packets on an outgoing scratch ring (`PACKET_RX`) in the order they are received. The `Route` microblocks take turns in an ordered fashion getting packets off the `PACKET_RX`. Once gotten, the packet is immediately prepared for an eventual multicast. `Route` then runs the GridStat filtering algorithm 2.1.1 on each outgoing port to determine which ports the packet is to be forwarded on. When a forwarding decision for each port has been made `Route` then iterates through all outgoing ports and forward packets on those meant to have the packet. Forwarding packets from the perspective of `Route` means putting the packet on a scratch ring corresponding to the link it's meant for. Packets prepared for a link but not meant to be forwarded should be dropped by `Route` but are allowed to pass through anyhow in order that performance may be more easily measured after simulation. The `Ethernet Transmit` (`TX`) microblock handles as many scratch rings are there are hardware links on the network processor hardware. `TX` will get packets from a ring then do what it must to send this packet out on the physical medium corresponding to the scratch ring the packet was found on.

3.3.2 Packet Receive (RX) microblock

This software component is a driver running on a single microengine. Although configurable to run on two microengines if the hardware line capacity demands it, something we will not discuss here as we currently have no such demand. RX is responsible for queuing all packets that arrive at any physical link on a single scratch ring (`PACKET_RX`). RX maintains logical packet ordering by making sure that all packets arriving on the same physical link are queued on `PACKET_RX` in the order they arrived on that link. However, order between packets arriving on separate physical links need not be maintained, and it's up to RX to choose which packet to queue on `PACKET_RX` first.

Input

RX is responsible for getting packets from the physical medium and into the software packet pipeline. This microblock is part of the IXA written by Intel and it handles all the configuration setup needed to initialize the hardware and logic for pulling packets off the physical media.

Output

All incoming packets are put on the `PACKET_RX` scratch ring.

Concurrency

RX can be configured to run on 1 or 2 microengines. We use the configuration of 1 microengine in our project.

Packet ordering

Packet order is maintained on a per link basis. So any number of packets arriving on the same physical link will remain their relative packet order when the packets are output on `PACKET_RX`. Packets arriving on different links have no ordering guarantees between them.

Performance

RX can keep up with line rates on all IXP hardware it's bundled with given proper configuration. Hence its performance is at least as good as we need.

3.3.3 Packet Transmit (TX) microblock

TX handles N scratch rings and N outgoing physical links. Each scratch ring has a fixed outgoing link that it corresponds to. TX is responsible for sending packets from all scratch rings out on its corresponding link in the order packets were queued on the scratch ring. TX maintains packet order within one scratch ring to physical link stream, and uses a simple round robin scheduling for the inter queue packet selection. TX can be configured with other scheduling algorithms like weighted round robin and negative deficits, but this is not important for the purpose of this project.

Input

TX has a number of scratch rings where packets are pulled as input. Each of these scratch rings correspond to an outgoing physical link in a 1 to 1 fashion. Packets are pulled from the scratch rings round robin.

Output

TX send packets out on the physical links of the network processor. The logic of TX is built to control the hardware needed to send packets that reside in DRAM out on the physical network medium.

Concurrency

TX can be configured to run on 1 or 2 microengines. We use the configuration of 1 microengine in our project. It utilizes network processor hardware optimized for transferring data directly from DRAM into the transfer buffers used by the hardware to generate packet data on the physical medium.

Packet ordering

Packet order is maintained per scratch ring. So any number of packets pulled from the same scratch ring will remain their relative packet order when the packets are output on the corresponding physical link. Packets pulled from different scratch rings have no ordering guarantees, however a simple round robin policy is used in this project's configuration.

Performance

TX can keep up with line rates on all IXP hardware it's bundled with given proper configuration. Hence its performance is at least as good as we need.

3.3.4 `Route` microblock

`Route` implements the GridStat routing and rate-filtering algorithm from sectino 2.1.1, and is the software component which determines whether and where a packet is to be forwarded. This microblock get packets from the `PACKET_RX` scratch ring and put packets on one or more of the `TX`'s scratch rings.

Input

Packets for route processing are pulled from the `PACKET_RX` scratch ring.

Output

Depending on whether and where a packet is to be forwarded, `Route` put each packet on the scratch ring corresponding to which outgoing link it belongs. If a packet is not to be forwarded anywhere, it should simply be dropped, but is allowed to pass through the router so that its performance can be measured from simulation packet logs.

Concurrency

`Route` can run on 1, 2, 3, or 4 microengines.

Packet ordering

Packet ordering is inherit when using the ordered threading model since the order in which threads do scratch ring put and get operations (interface between microblocks) is always the same. E.g. Thread 1 gets a packet, then thread 2 gets a packet, and etc. until thread 8 gets a packet. Then Thread 1 puts a packet, then thread 2 puts a packet, and etc. until thread 8 puts a packet. And only when that entire sequence is complete it start again with thread 1 getting another packet.

Routing algorithm

The `Route` microblock implements the algorithm described in section 2.1.1 After getting a packet P from the `PACKET_RX` scratch ring, `Route` makes a meta-data and reference copy of P for each link except the one P came from and one other where P itself will be used. It's not a full packet copy, since only the meta data is copied. In a production implementation this should be changed to use proper reference counting for multicast support. In the configuration used here, there are only 3 links, hence only 1 copy is needed. e.g. If P arrived at link 0, P would be the packet to send out on link 1, and the one copy of P would be the packet to send out on link 2. Even though packets are copied and prepared for sending this early on in the algorithm, a forwarding decision must still be made on a per link basis. After starting a packet pseudo copy, the packet data needed to decide whether and where it's to be forwarded is downloaded from the packet buffer in DRAM. For GridStat this is currently the variable-ID $P.varid$ and time-stamp $P.timestamp$, a total of 12 bytes which can be extracted in a single DRAM read operation. After initializing some registers used to store forwarding decisions, preparing packet copy to link assignment, and getting all needed packet data from DRAM, execution enters a loop which iterates through all links (skipping the link where P arrived). For each link L, we call the `dl_gridstat_filter` macro, which will decide whether P is to be forwarded on L. The per link decision is marked on a bit-array, where a set bit signifies a decision to forward the packet on the corresponding link. When the loop has completed, the result is the updated decision bit-array. `Route` then enters a final loop iterating through all links sending or dropping packets according to the decision bit-array. After iterating through all links and forwarding/dropping packets, the algorithm starts over by getting another packet from the `PACKET_RX)` scratch ring.

The `dl_gridstat_filter` macro uses a link number, and the following fields of P as input: variable-ID, time-stamp, publication rate. As output a single register called `forward` that is given value false if the packet is to be dropped, and true if the packet is to be forwarded. First in the macro we ensure that the routing table entry of *P.variableid* is in local memory, then we perform a filtering operation and use the truth value to determine whether P is to be forwarded. The operation is as follows: $(P.timestamp + P.pubint/2)$ *modulo divisor*$) < P.pubint$, and corresponds to lines 8 and 9 of the routing algorithm in section 2.1.1. The *modulo* operator is implemented with the `remainder64x32` macro as the IXP hardware does not provide any general division or modular operator. The macro does not implement general integer division or modulo in software either, rather it uses reciprocal multiplication to compute the 32-bit remainder of a 64-bit dividend which is exactly what the algorithm requires. Reciprocal multiplication exploits some properties of manipulating numbers that are represented in binary, i.e. the fact that $x/2^n = x >> n$ and $x * 2^n = x << n$. In order to understand how reciprocal multiplication works, consider the following. By definition:

$$Remainder = Dividend \ modulo \ Divisor$$

and

$$Dividend = Quotient * Divisor + Remainder$$

If we only need to find the *Quotient* (like in division), then the value of *Remainder* does not matter and we may assume that $Remainder = 0$ allowing the previous equation to transform into:

$$Quotient = \frac{Dividend * \frac{2^n}{Divisor}}{2^n}$$

Let R_n be the n-bit reciprocal $\frac{2^n}{Divisor}$ of *Divisor*. Appropriate values for n and R_n are computed based on *Divisor* before stored in the routing table, see the `computeReciprocal` method in A.2 for details on how to compute them. Given n and R_n from routing table lookup at runtime, the *Quotient* may be computed through only multiplication and shift operators as:

$$Quotient = (Dividend * R_n) >> n$$

and once the *Quotient* is known, then *Remainder* may be computed through only multiplication and subtraction operators as:

$$Remainder = Dividend - (Quotient * Divisor)$$

The value of the dividend is given by the relevant packet's time-stamp, the divisors are known in the algorithm's inner loop as the subscriber interval for which n and R_n are pre-computed and to be found in the routing table.

Collapsing subscriber rate list See figures 3.4 and 3.5 on how to collapse subscriber lists for the routing table.

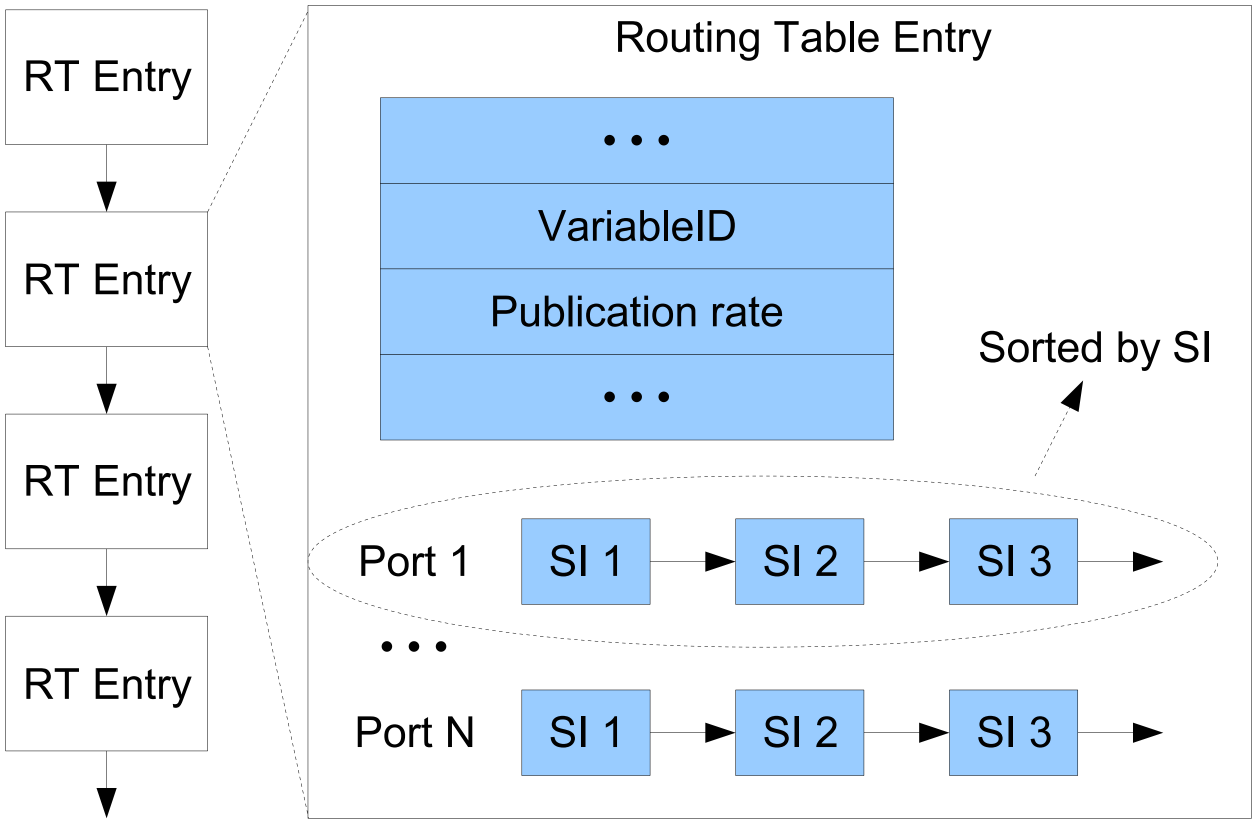

Figure 3.2: Routing table data structure

The routing table data structure consists of a single linked list of routing table entries. Each routing table entry has an associated variable-ID, publication rate, and a linked list of subscriber intervals for each port (link) of the hardware. A routing table entry can also be found by a hash lookup using the variable-ID as key. The IXP2xxx platform have hardware capable of doing very fast hash generation.

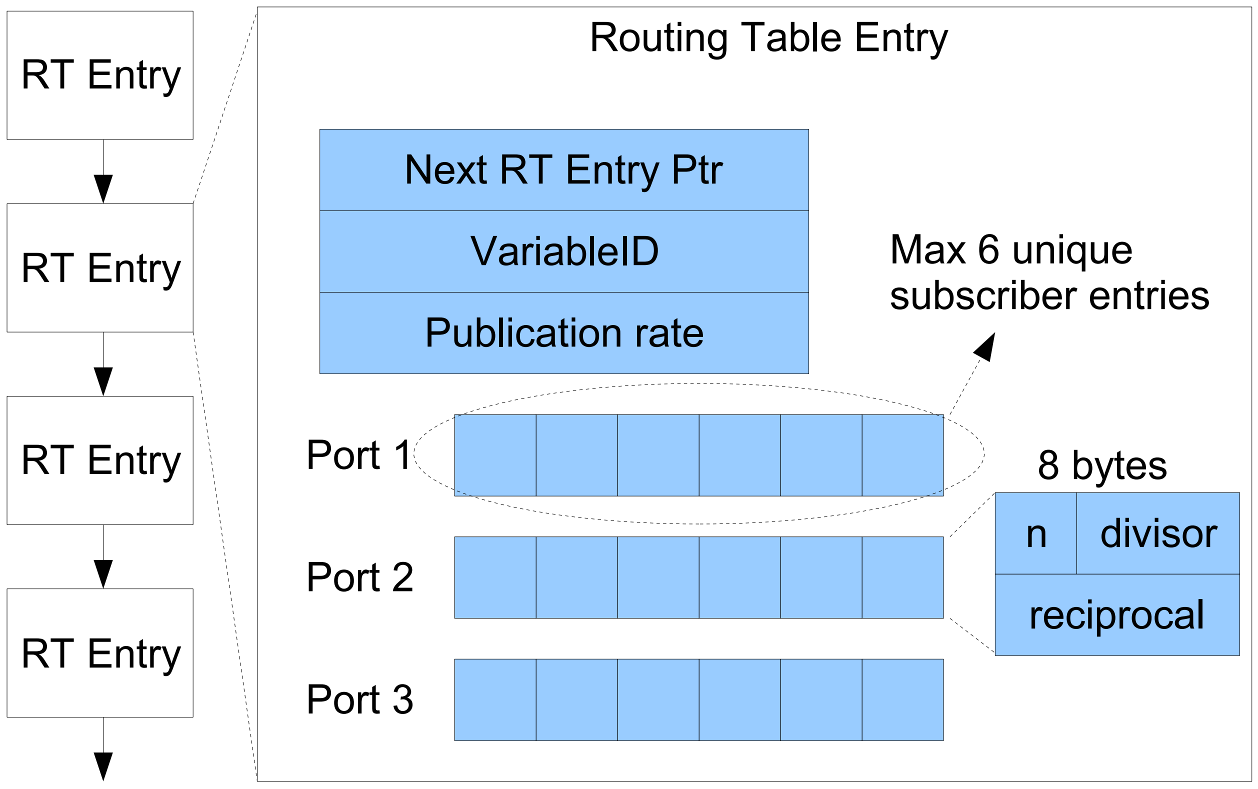

Figure 3.3: Routing table entry details

Dynamic data structures like linked lists are cumbersome and slow to work with from the data-plane. Microengines prefer data structures that are small and sequential in memory as to minimize memory read operations. Routing table entries therefore use a fixed size array for each port which contains a collapsed list of subscriber intervals.

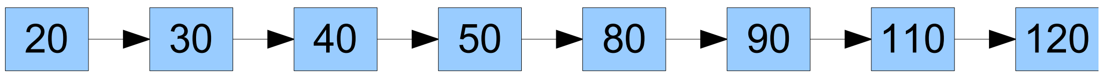

Figure 3.4: Original subscriber list

As an example consider the above list of subscriber intervals. Due to the modular arithmetic in the `dl_gridstat_filter` macro, we need only check a subset of the subscriber intervals. This subset is found by eliminating any subscriber interval value SI_i when there exist another subscriber interval of lower value SI_j such that $SI_i \ MOD \ SI_j = 0$ holds.

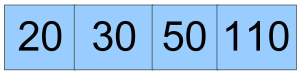

Figure 3.5: Collapsed subscriber list

Above we see the list of subscriber intervals after elimination. Note how 40, 80, 90, and 120 were eliminated due to $40\ MOD\ 20 = 0$, $80\ MOD\ 20 = 0$, $90\ MOD\ 30 = 0$, $120\ MOD\ 20 = 0$.

Cache `Route` gets routing table entry information from a cache that resides in microengine local memory. If the cache does not hold the desired routing table entry, it's downloaded to the local memory cache from SRAM. Only one routing table entry is ever used by a single packet, and the same entry is often used very frequently across packets. The IXP2400 has a per-microengine Content Addressable Memory (CAM) providing a fast and easy to use 16 entry cache.

We can think of the CAM as a 16 row table, where each row has the following fields: A 32-bit tag value that it written by the programmer. A 4-bit state field also written by the programmer. An entry of the time-ordered Least Recently Used (LRU) list. The LRU list keeps track of which row was most and least recently used at all times. Using the CAM is simple and fast. In order to use the CAM as a cache in the `Route` microblock we do as follows. The `cam_clear` instruction is called once per microblock running `Route` upon initialization. Whenever a packet P is processed by one of the 8 threads in a microengine, before we enter the loop iterating through all links calling `dl_gridstat_filter`, a `cam_lookup` on $P.variableid$ is performed to check whether the current microengine has P's routing table entry in local memory (cache). If we get a cache hit (the routing table entry is present in

Entry 0	LRU list	4-bit state	32-bit tag
Entry 1	LRU list	4-bit state	32-bit tag
Entry 2	LRU list	4-bit state	32-bit tag
Entry 3	LRU list	4-bit state	32-bit tag
Entry 4	LRU list	4-bit state	32-bit tag
Entry 5	LRU list	4-bit state	32-bit tag
Entry 6	LRU list	4-bit state	32-bit tag
Entry 7	LRU list	4-bit state	32-bit tag
Entry 8	LRU list	4-bit state	32-bit tag
Entry 9	LRU list	4-bit state	32-bit tag
Entry 10	LRU list	4-bit state	32-bit tag
Entry 11	LRU list	4-bit state	32-bit tag
Entry 12	LRU list	4-bit state	32-bit tag
Entry 13	LRU list	4-bit state	32-bit tag
Entry 14	LRU list	4-bit state	32-bit tag
Entry 15	LRU list	4-bit state	32-bit tag

Figure 3.6: CAM structure

Each microengine has it's own local Content Addressable Memory. The IXP hardware provide very fast CAM compare and lookup instruction with automatic LRU list updates making the CAM the perfect cache support mechanism.

local memory), then nothing is explicitly done to update the CAM but the hardware will mark the entry that matched as Most Recently Used (MRU). If we get a cache miss, then the routing table entry must be downloaded from SRAM to local memory before it can be used. Before the download process starts the CAM tag field of the LRU item is written with the variable-ID of the relevant routing table entry. However, the state field is marked with a value indicating it being invalid, so that if another thread T_x try a lookup for the same variable-ID before the routing table entry is downloaded from SRAM, T_x will get a cache hit but with associated state value indicating that it's being downloaded. The `dl_gridstat_filter` macro being called inside the loop iterating through all links, performs a second `cam_lookup`. This will always produce a cache hit, but the data might be either valid or invalid (being downloaded). If the data is valid the macro continues the filtering process using that data, If the data is invalid then T_x will yield (force context switch) and try the lookup again until the result of the lookup is a hit with valid data. A cache miss is structurally impossible for T_x since there are fewer other packets processed on this microengine (at most 7) before P gets processed than there are entries available in the CAM, hence the entry which has the matching variable-ID but invalid data will never become the LRU item (the LRU item is the item to always overwrite when cache is full) while P is being processed.

Performance

Several issues make good performance hard to achieve in the `Route` microblock. Accessing DRAM for packet data extraction, accessing the SRAM for routing table entry lookups.

Performance of the routing algorithm itself is also very important. There are four kinds of RAM access done within `Route`. Scratch, DRAM, SRAM, and local memory. Scratch access is used getting packets from the `PACKET_RX` scratch ring, and putting packets on the transmit rings. Local memory access is used by the routing table entry cache and is discussed in a paragraph below. DRAM and SRAM access are both discussed in their own separate paragraphs below.

Routing algorithm Performance on the routing algorithm itself depends on several variables. Most importantly it depends on how many entries in the subinterval list that need be checked. There are two nested loops, the outer loop iterates through all outgoing links, and the inner loop iterates through all subintervals in the routing table entry lookup table. The outer loop need worst case consider all outgoing links (except 1) as they all have different lookup tables. The inner loop has several complications to overcome. It uses modular arithmetic for every subinterval in the subinterval list to decide whether the packet can be forwarded. A separate macro `remainder64x32` implements the modular operation. This macro does at least 60 instructions making it very expensive.

DRAM access DRAM access, this is only to extract information about the packet needed by the routing logic. It's a single read-only access done once per packet pulled from the `PACKET_RX` scratch ring. DRAM lookup is never needed on copies of packets, as everything (except destination address) about the copy is the same as the original.

SRAM access SRAM access is used to look up routing table entries.

Cache The cache in `Route` is used to store routing table entries that have been recently used by the microengine running `Route`. On a cache hit we see that routing table entry lookup is very fast, reading from local memory takes at most 3 cycles per long word whereas a register read takes 1 cycle. On a cache miss we download the routing table entry from SRAM, which takes takes 3 SRAM read operations of 8 long words each which implies waiting on IO.

3.4 IXP2xxx synchronization and locking mechanisms

This prototype implementation does not use synchronization mechanisms other than the atomic scratch ring put and get and sram queue and dequeue instructions. Those instructions automatically handle synchronization, and so the microcode of the `Route` microblock has no critical regions from a programmer's point of view. The hardware does however support many different synchronization mechanisms, of which some probably must be used in order to implement features like routing table updates at runtime. In particular, the sram and scratchpad memory have supporting atomic operations to do read-modify-write instructions that are guaranteed to appear atomic to all other atomic operations (but not to non-atomic operations). We already mentioned the special ring and queue instructions which are used in this implementation, but there are also general atomic operations like test-and-set, test-and-clear, test-and-incr, test-and-decr, test-and-add, swap, and a few others. These instructions are more than sufficient to implement any kind of synchronization between microengines, but care must be taken so that synchronization contention does not prevent microengines from achieving expected degree of parallelism.

3.5 Constraints on the IXP2xxx hardware

3.5.1 Soft constraints

Division and Modular operators

The Intel IXP 2400 and 2800 series of hardware has no hardware support providing division or modular (remainder) operation. So these operations must either be implemented in software, or avoided. The GridStat router algorithm requires modular computation (remainder operator) where all the divisors are known in advance. The straightforward solution of implementing a remainder function in software suffers from very poor performance. As the amount of remainder operations needed per packet can be very many, then even at traffic rates at far less than half of line capacity the software would be unable to keep up with traffic rates resulting in massive packet drops once the packet pipeline buffers (scratch rings) fill up. A small but significant improvement over using general software division implementations is to use precomputed divisor reciprocals to speed up the remainder function. See section 3.3.4 for details.

Precomputed reciprocals The solution using precomputed reciprocals requires extra work at the routing table setup and manipulation stages. In the implementation example we only do this work at routing table setup as we omit changing the routing table at runtime for simplicity.

3.5.2 Hard constraints

Microengines

The IXP2400 has 8 microengines. A minimum of 2 of them are always used to control the physical receive and transmit hardware, and are usually occupied by the `Ethernet Receive` and `Ethernet Transmit` microblocks provided by the IXA. This leaves 6 microengines to use in the application. As demonstrated by applications like IP Forwarding, Diffserv, and others, this suffice for most applications needing typical IXP2400 line speed throughput. The GridStat routing algorithm (see section 2.1.1) have one major scalability challenge, it allows many unique subscriber intervals (no common factors except the value 1) per status variable. This challenge cannot be completely overcome by adding more microengines to the hardware, as the time complexity of the algorithm grows much faster than any reasonable hardware upgrade in amount and speed of microengines could mitigate. With a frequency of 600 MHz the microengines should not spend more than about 100 cycles per packet in order to keep up with line rates of 2-3Gbps. Applications that need more instructions than this must run their microblock on several microengines in parallel to mitigate this. However, as the amount of microengines increase, the contention of critical regions get worse and requires careful design and implementation to fully utilize all CPUs which isn't always possible.

Memory

The 256MB of DRAM currently available on typical IXP2400 is by far sufficient in order to implement a complete GridStat status router. Part of the DRAM is used as system memory of the local network processor OS, the rest is used to hold packet data. The SRAM of 64MB should suffice for routing tables in the GridStat project with proper data structure design. If we assume routing table entries of 64 byte, this adds up to a routing table capacity of one million flows. The scratch memory area of 8MB is used only for scratch rings in this project and the size is plenty sufficient. Microengine local memory of 640 long words of 4 bytes each suffice for a medium size (16 entries) cache. The microengine CAM only has 16 entries on the IXP2400 hardware, which is small but could be utilized for larger groups like routing table pages and not only status variable ids.

3.6 Features not included in the status router prototype

In order that the network processor prototype described in this chapter may become a full-featured GridStat status router, the following functionality must be added:

- Add ability for the GridStat management-plane to interface with the status router at runtime. This is generally about updating the SRAM routing table as publishers are added/removed and subscribers add/remove or change their subscriptions or as relevant network topology changes. This would also require proper routing table synchronization between management-plane and the microengines who share the routing table data structure.

- Mode switching must be implemented, with appropriate interface for controlling it either via management packages arriving on the data-path or other management messages arriving through host computer PCI bus or Ethernet debug interface.

- Support for packets with more than one status variable contained in it and/or more than one sample of the same status variable in the same packet. Currently, the prototype expects only packets with a single sample of a single status variable.

- The mapping of variable-IDs to routing table entry SRAM-addresses should be done through hash lookups rather than be hard-coded in order to support numerous values. This would also require a change in the routing table initialization script.

- The routing table must be used for publish interval value lookups, rather than the hard-coded value of 10 millisecond used. This is a very simple task.

Chapter 4

Analysis and Simulations

This chapter will demonstrate how the GridStat status routing algorithm (see section 2.1.1) may perform on an IXP2400 chip. Analysis form the basis of expected and ideal performance while simulation is used to estimate real performance. The SDK used to implement the status router includes a simulator to run microengine code, this is a true simulator that interprets the same compiled code that would be used with real hardware. The simulator is deterministic and cycle accurate, which means that running microengine code in the simulator is predictable and will produce results as if it was run in real hardware, but in slow motion. The simulator is most easily used with fixed rate incoming packet streams, hence testing many different throughputs requires many different simulations. Each experiment in this chapter consists of several independent simulations where hardware configuration, code, and scripts remain the same while the throughput of the incoming packet stream varies from one simulation to another. The throughput capacity and expected packet processing times are estimated for each experiment and then compared across experiments in a summary. The analysis and simulation results also reveal some scalability properties that are summarized in the end.

4.1 Experiment configuration details

Two configurable aspects of simulation determine the outcome of the experiments:

- The stream of packets arriving at the various ports, or "input" for short. The definition of input also includes a simulation configuration option of the packet arrival rate for each port measured in Mbps. Note that the actual timestamps in the status variables described by packets are synthetic and that the difference in timestamps from one packet to the next do not reflect the time passed between their respective packet arrival. The input data is on the GridStat packet format where the variables relevant to routing are the status variable id and the publish time-stamp of the packet. These variables must be inspected for each packet at runtime in order to determine whether and where a packet is to be forwarded. In a single simulation run, the input stream is a fixed rate, so in order to see the performance of different throughputs the simulation is rerun within the same experiment with different input rates. Variable (non-fixed) input rates are not analyzed or simulated in these experiments. The packet data stream is a file generated by a Java program (see code listing A.1) with a set of configurable options. The bytes of the packet data stream used for all experiments are created by invoking the following parameters to the PacketStreamGenerator:

  - “experiments-p0 1 500 244813791234”
  - “experiments-p1 2 500 244813667203”
  - “experiments-p2 3 500 244813641339”

- The routing table. The routing table holds for each status variable and outgoing port a collapsed list (see 3.3.4 for details) of subscriber intervals (i.e. the computed reciprocals of the intervals). The routing table setup script is generated by a Java program (see code listing A.2) with a set of configurable options. The routing table is kept exactly the same across all simulation runs within a single experiment, and is never the same across experiments.

Other aspects that could significantly change the result of the experiments are kept the same across all experiments. These aspects include:

- The routing algorithm used. One single algorithm implementation is used.
- Microengine code other than the routing algorithm.
- Hardware simulation configuration with the exception of throttling input rates done by changing incoming line speed capacity. This is set to be an IXP2400 B0 chip with a microengine clock frequency of 400MHz. The incoming and outgoing network devices is set to be a x32MPHY4 with 3 ports each and the external media bus clock runs at 104MHz.
- Scripts for simulation setup and SRAM/DRAM memory layout, with the exception

of routing table setup in SRAM. This includes things like packet buffers and packet meta-data memory area.

A simulation configuration is a specific combination of hardware configuration, initialization scripts (includes routing table setup), microengine code, and input data streams. These parameters control the runtime behavior and output of the simulated network processor and suffice to allow the simulator to deterministically process the input and produce output in the form of outgoing packet streams. The simulator is deterministic, which means that every time the same simulation configuration is simulated it will always produce the exact same result. A simulation run, means the simulator running a specific simulation configuration starting with hardware initialization and running some number of microengine clock cycles while input is injected through one of the simulators input devices. Exactly how many microengine clock cycles to run is either controlled manually or by predefining some number of packets that should be injected or transmitted by the simulator.

Simulations are generally run so that at least 1000 incoming packets have completed processing. However, in some simulation runs where the incoming throughput is either very low or very high compared with the capacity of the router, the results are obvious after much fewer packets have been processed and such simulations may be stopped before 1000 packets have been processed.

4.2 Performance measurements

Performance of networks in general and also routers are chiefly measured through three measurements with sometimes very dynamic properties. They are throughput, latency, and jitter. Properties of throughput receive particularly much attention here because it relates so strongly to scalability of the GridStat algorithm. Throughput, latency and jitter challenges can often be satisfied with hardware and/or software improvements because most routing applications have well understood algorithms with performance relatively independent of packet content. The GridStat routing challenge is special because the routing algorithm itself has runtime dependencies to a management plane (routing table) and packet stream data contents that could significantly affect its performance and scalability properties.

Statistics of packet processing time (delay) at different throughputs are key when measuring performance of these experiments. In general the throughput capacity of a network processor router is dependent on runtime features like complexity of incoming packets, routing table management and current internal thread utilization. These experiments are designed so that the current internal thread utilization is the only significant runtime aspect that is directly related to the throughput capacity. Statistics like mean, minimum, maximum, and standard deviation of the packet delays are relevant only as long as the input rate does not exceed the throughput capacity of the router. Each experiment attempt to estimate the router's throughput capacity by the same input stream at different throughputs and observing the dynamics of packet processing times. Each experiment estimates a routing table runtime complexity for its input when processing exactly 1000 packets.

In general the routing table throughput capacity is said to be exceeded if the incoming packet stream enters faster than the router is capable of processing, which must eventually lead to unwanted packet drops. A method of estimating the routing table throughput capacity is needed. Let $Delay(s,t)$ be a function so that its output is the expected processing time of a packet that enters the router at time t and where s represents a simulation run. Delay samples are observed by analyzing the output logs of the simulation run of s, and form the data set used to estimate $Delay(s,t)$ through a least squares linear regression model. $Delay(s,t)$ is on the linear form $Delay(s,t) = intercept + (slope * t)$. We say that the incoming throughput rate of a simulation exceeds the router's capacity if the *slope* part of the corresponding $Delay(s,t)$ is greater than 0. Approximating $Delay(s,t)$ is done by a Java program (See code listing A.5) that inspects the packet logs after running s.

Packet receive and transmit "start of packet" (SOP) and "end of packet" (EOP) is measured in units of media bus cycles since beginning of simulation, simply referred to as cycle X. Packet processing time is also measured in media bus cycles, referred to as cycles. Packet processing time is the difference between packet transmit SOP and packet receive EOP, in other words the time it spends in the actual packet processing software pipeline along the fast data path. Note that a certain number of media bus cycles do not reflect the number of instructions that the microengines carry out, since media bus cycles are much slower (about 4 times) than microengine cycles and some instructions take more than one microengine cycle to carry out due to internal IO latency.

For a single simulation the following statistics are compiled by analyzing its packet logs and shown through tables and graphs of the experiments:

- Number of packets entering and leaving the system.
- Inter packet arrival time (in cycles). A mean measurement with a standard deviation generally so low that it can be thought of as an exact measurement. In practice this means we have a fixed rate input stream. However, the simulator will start sending packets right away even before the network processor has been initialized, this will cause the observed inter packet arrival time to be smaller at the beginning of a simulation then quickly even out to a fixed rate.
- Standard deviation of inter packet arrival time (in cycles). This give a handle on the jitter added to the packet stream passing through the router.
- Mean, minimum, maximum and standard deviation of the observations used to estimate $Delay(s,t)$ (in cycles). This is only relevant as long as the simulation does not exceed the router's throughput capacity.

4.3 Experiments overview and expectations

A total of 5 experiments is conducted where the only significant change in simulation configuration between any two experiments is the routing table. When we consider a packet being processed by the routing algorithm, the combined length of the collapsed subscriber lists of the packet's corresponding status variable and target ports define the algorithm's processing time for that packet. In fact these subscriber list lengths (i.e. routing table contents) is the only free variable in the routing algorithm and hence the variable that defines the algorithm's asymptotic time complexity. More than one such subscriber lists are used (due to multicasting) by the algorithm when processing a packet, and part of the payload data (status variable id and time-stamp) of the packets decide how many of the elements in each relevant subscriber list need be considered. Hence it's impossible to determine the average number of iterations of the inner loop of the routing algorithm until the routing table and input packet stream is known. All experiments use the same pre-generated packet stream as input, and only the first 1000 packets passing through is considered in performance evaluation.

Let e be an experiment. Let $RoutingTable$ be a function of e such that $RoutingTable(e)$ defines the routing table of e. Let $RuntimeComplexity$ be a function of routing tables such that $RuntimeComplexity(RoutingTable(e))$ defines the runtime complexity of the routing table of e. $RuntimeComplexity(RoutingTable(e))$ is the per packet average number of routing table lookups required to route the first 1000 packets from the combined input streams of e. Table 4.1 maps e to $RuntimeComplexity(RoutingTable(e))$ which corresponds to the number of inner loop iterations of the algorithm:

Let K_1 be the minimum cost of routing a packet (i.e. when there are no subscribers for the corresponding status variable). Let K_2 be the cost added per iteration of the algorithm's inner loop.

Let $ThroughputCapacity$ be a function of $RuntimeComplexity$ such that

Table 4.1: Mapping e to $RuntimeComplexity(RoutingTable(e))$

e	$runtimecomplexity$
1	0
2	2
3	7.08
4	14.16
5	21.24

$ThroughputCapacity(RuntimeComplexity(RoutingTable(e)))$ defines the throughput capacity of experiment e measured in megabits per second (Mbps). Intuitively the throughput formula to find the packets per second capacity is:

$$pps_{\frac{packets}{second}} = \frac{computationcapacity_{\frac{instructions}{second}}}{packetcost_{\frac{instructions}{packet}}}$$

The relationship between packets per second and megabits per second is:

$$throughput_{\frac{megabits}{second}} = \frac{pps_{\frac{packets}{second}} * bpp_{\frac{bits}{packet}}}{10^6}$$

Furthermore

$$packetcost_{\frac{instructions}{packet}} = K_{1instructions} + K_{2instructions} * RuntimeComplexity(RoutingTable(e))$$

and

$$computationcapacity_{\frac{instructions}{second}} = me_{microengines} * clock_{\frac{\frac{tick}{second}}{microengine}} * ipt_{\frac{instructions}{tick}}$$

where me is the number of microengines running the routing algorithm in parallel.

Consider the three following assumptions:

Assumption (1): All internal IO of microengines is successfully hidden through microengine concurrency.

Assumption (2): Each instruction take exactly 1 clock cycle to complete, implying $ipt = 1$.

Assumption (3): Running the algorithm in me parallel microengines (where $me > 1$) is me times faster than running the algorithm with only 1 microengine.

By assuming that (1), (2), and (3) holds, the expected throughput capacity is upper bound by the following function:

$$IdealThroughputCapacity(x) = \frac{me * clock * ipt * bpp}{10^6 * (K_1 + K_2 * x)}$$

where x is the number of iterations of the inner loop of the algorithm.

In these experiments the following applies:

K_1 and K_2 can be estimated by counting the number of instructions in the microengine code that implements the routing algorithm. A minimal (thereby ideal) instruction path of a single packet through our implementation yield $K_1 \geq 320$ and $K_2 \geq 88$.

The microengine clock is set to 400Mhz for all simulations.

The ideal throughput capacity (in Mbps) function for these experiments is therefore:

$$IdealThroughputCapacity(x) = \frac{me * clock * ipt * bpp}{10^6 * (K_1 + K_2 * x)} = \frac{2 * 400 * 10^6 * 1 * 528}{10^6 * (320 + 88 * x)} = \frac{422400}{320 + 88 * x}$$

Figure 4.1 shows the $IdealThroughputCapacity(x)$ function drawn as a graph for x such that $0 \leq x \leq 30$.

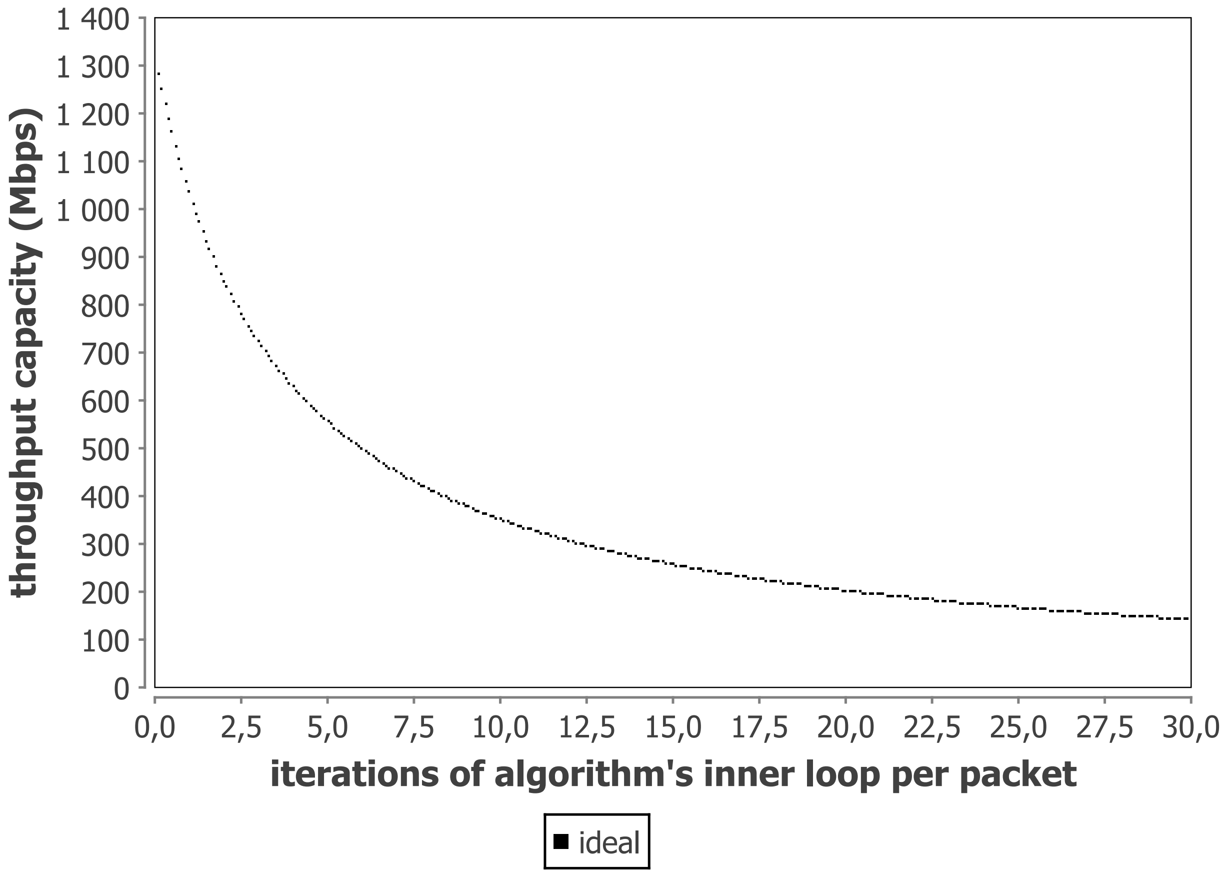

Figure 4.1: $IdealThroughputCapacity(x)$

A realistic estimation of the throughput capacity would not assume that any of (1), (2), or (3) holds. Though network processors are designed to get close to these ideals, it does take careful implementation of code. Such careful implementation is outside the scope of this thesis, so it must be expected that the implementation used here suffer a significant penalty compared with a more ideal implementation.

For our prototype, let F be the ideal throughput capacity function and let F' be the real throughput capacity function. F' is expected to have the same general shape as F and $F'(x) < F(x)$ is expected to hold for all positive values of x. Exactly how great the difference between F and F' is best explored by estimating F' through simulation.

4.4 Experiment 1

4.4.1 Setup

Routing table with no subscribers hence a runtime complexity of 0 no matter what the input is. RTSetupCodeGenerator run with no parameters.

4.4.2 Analysis

This experiment estimates the real throughput capacity $TC(x)$ when $x = 0$. A more realistic K_1 can then be estimated by the equation:

$$K_1 = \frac{me * clock * ipt * bpp}{10^6 * TC(0)} = \frac{2 * 400 * 10^6 * 1 * 528}{10^6 * TC(0)} = \frac{422400}{TC(0)}$$

Observe that the above equation assumes that assumptions (2) and (3) still holds. Though this is not completely accurate, it suffices to quantify the difference between the ideal throughput capacity and an estimation of the real throughput capacity.

A baseline throughput capacity of 692 Mbps (see figure 4.3) might seem like a very low and bad capacity of hardware supposedly capable of almost 4 times this throughput. The reason for this low capacity is that the routing algorithm implementation is not at all optimized, nor designed to utilize the full potential of the network processor. Having such a "slow" implementation is in fact a benefit when attempting to uncover scalability properties of the algorithm. When the algorithm implementation performs way below the capabilities of every other part of the routers software and hardware even for simple inputs and routing tables, we can be sure that it will always be the bottleneck of throughput capacity. This makes every change affecting the routing algorithm's packet processing times in turn affect the observed total packet processing times proportionally. The algorithm as a slow bottleneck also makes the router very sensitive to variables like routing table runtime complexity which in turn becomes very visible in packet processing times.

4.5 Experiment 2

4.5.1 Setup

Routing table with a runtime complexity of exactly 2 no matter what the input is.

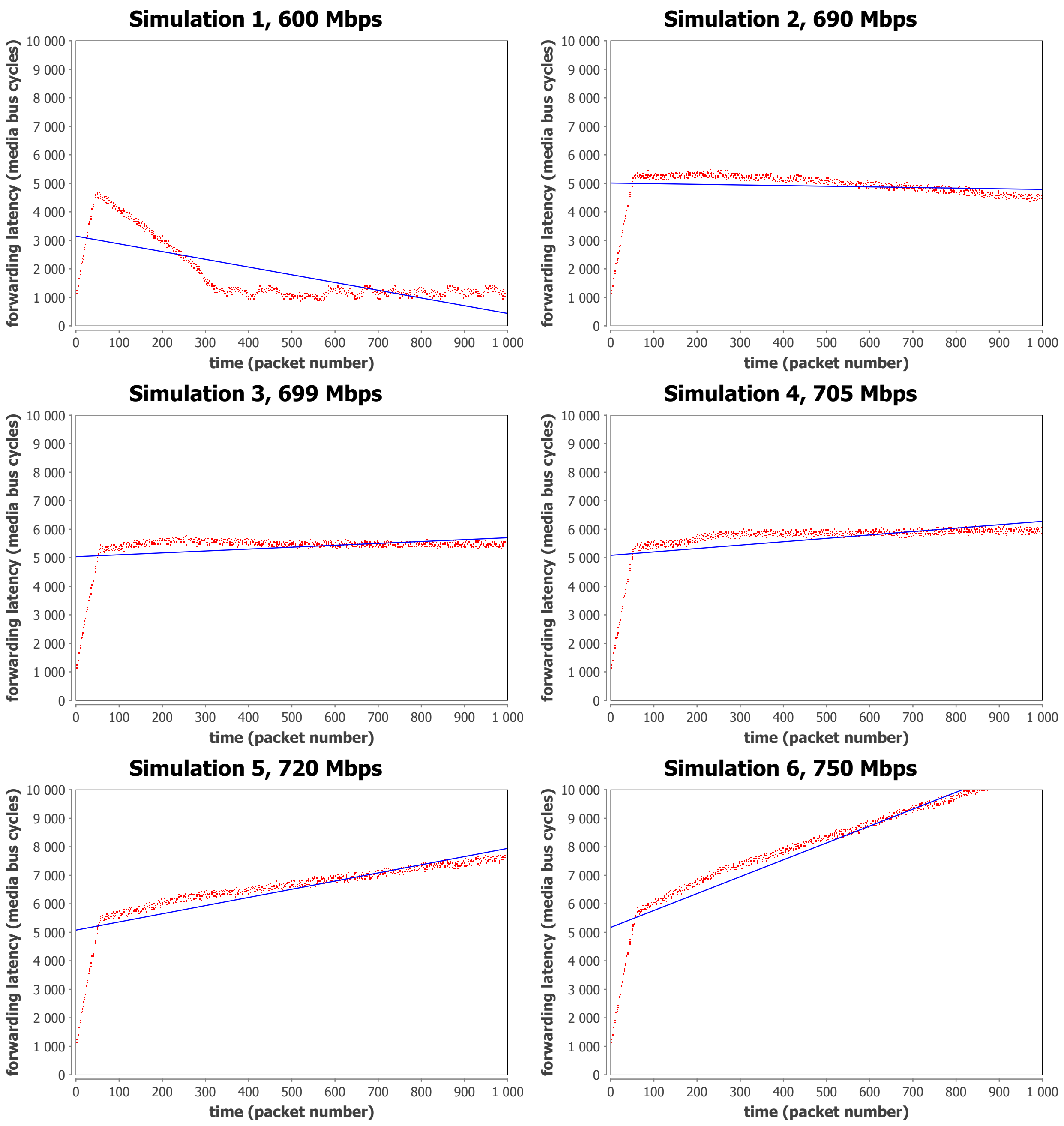

Figure 4.2: Simulations for experiment 1

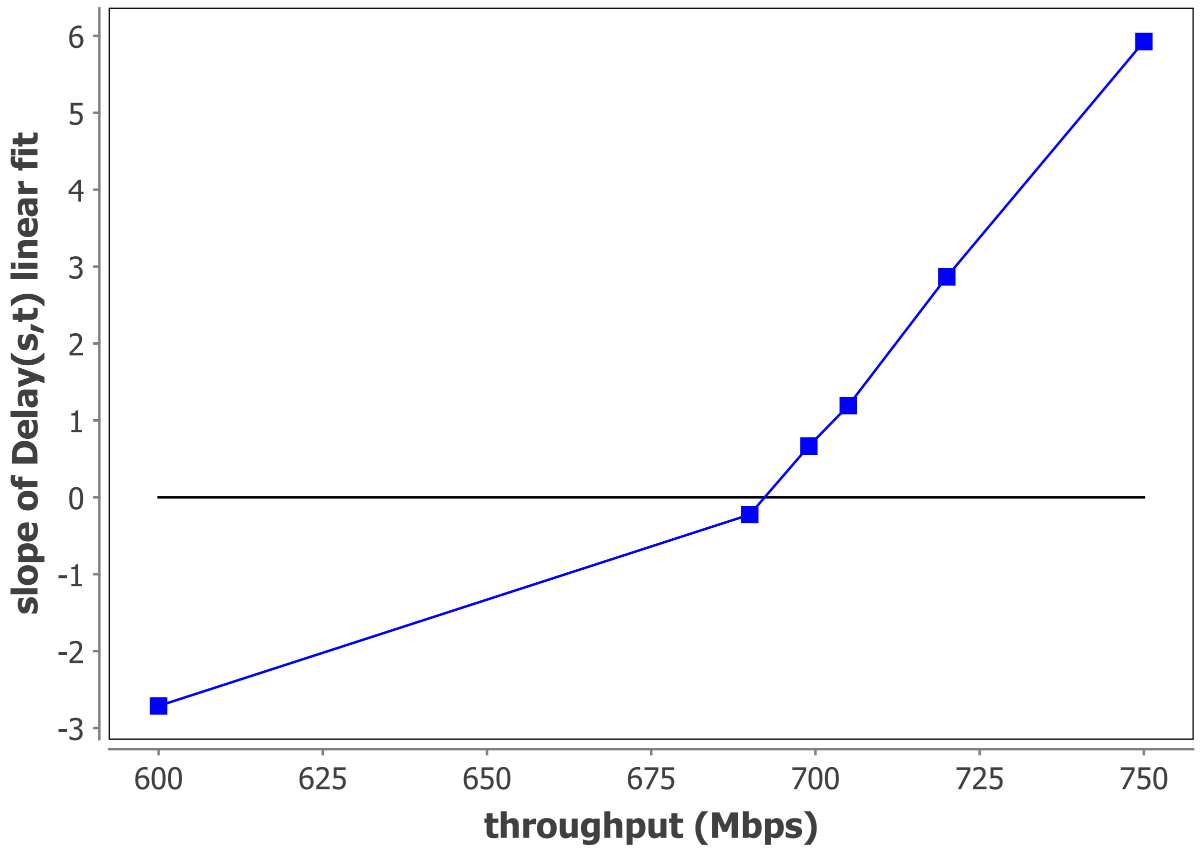

Figure 4.3: Slopes of experiment 1

Throughput capacity is approximately 692 Mbps.

Table 4.2: Incoming and outgoing inter-packet-delay per simulation of experiment 1

s	Input (Mbps)	Input Average Inter-packet delay (media-bus cycles)	StdDev	Output Average Inter-packet delay (media-bus cycles)	StdDev
1	600	274.713	0.452456	265.303	122.719
2	690	238.887	0.317176	236.583	112.661
3	699	235.807	0.395132	236.519	116.301
4	705	233.802	0.398545	235.637	69.2877
5	720	228.930	0.255289	235.676	122.151
6	750	219.773	0.418908	235.581	111.718

4.5.2 Analysis

The router has a throughput capacity of about 508 Mbps in this experiment, see figure 4.6. The inner loop is run exactly 2 times for every packet, and already there is a noticeable drop in throughput capacity. This experiment represents the "cheapest" routing we expect to get, even if subscribers were strictly constrained in their subscriber intervals.

4.6 Experiment 3

4.6.1 Setup

Routing table with a runtime complexity of 7.08 for the first 1000 packets of the data stream.

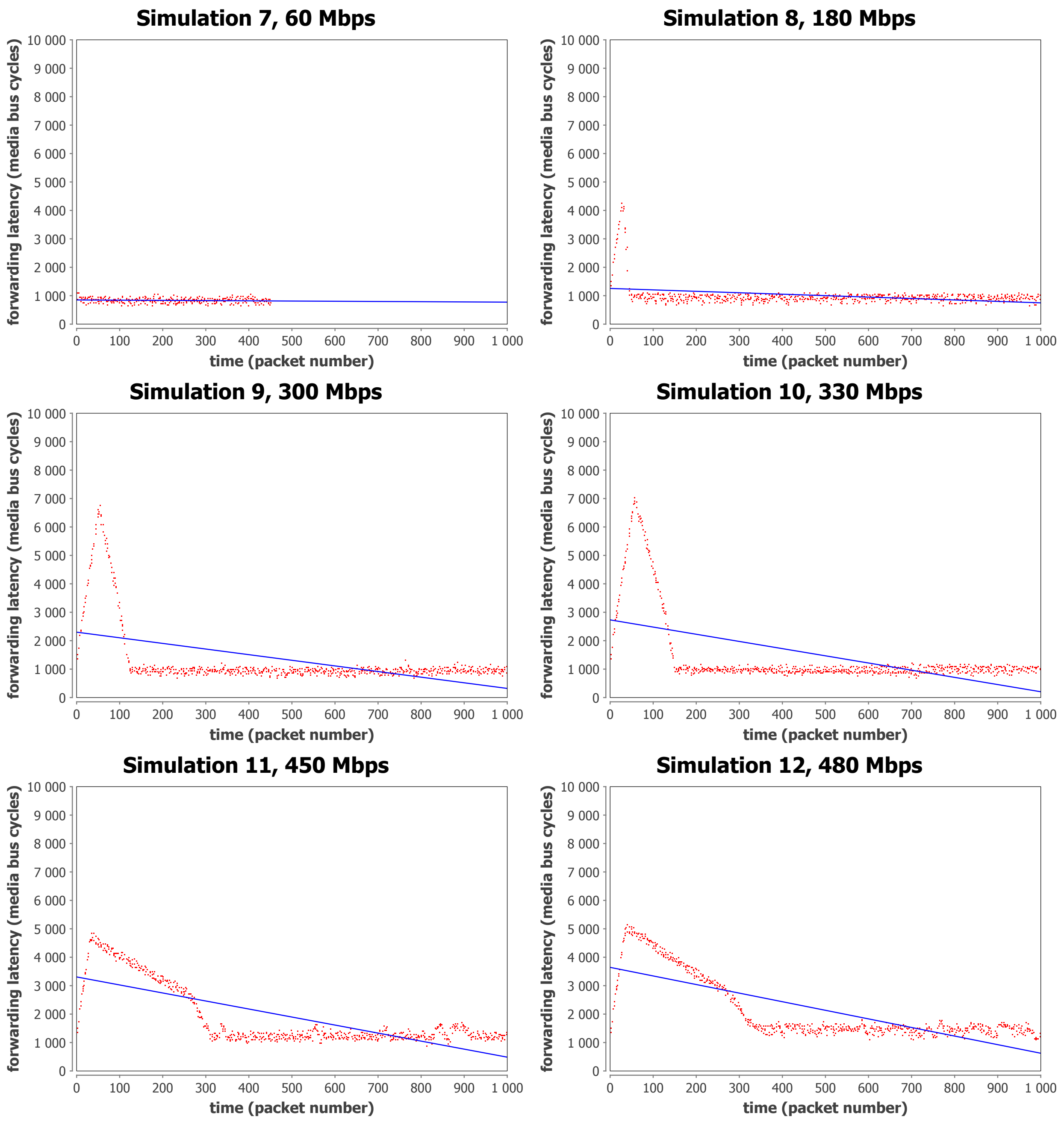

Figure 4.4: Simulations for experiment 2 part 1

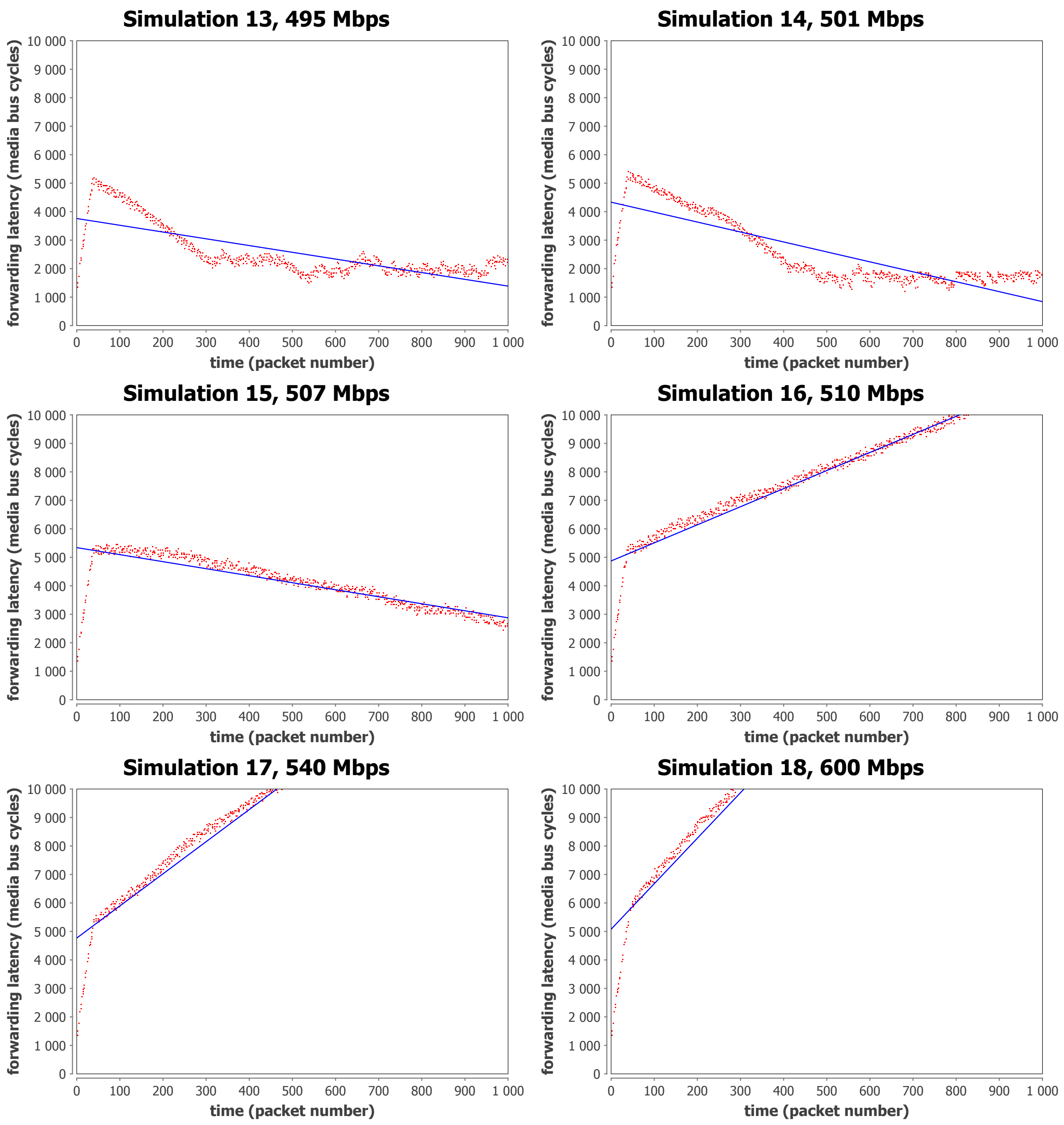

Figure 4.5: Simulations for experiment 2 part 2

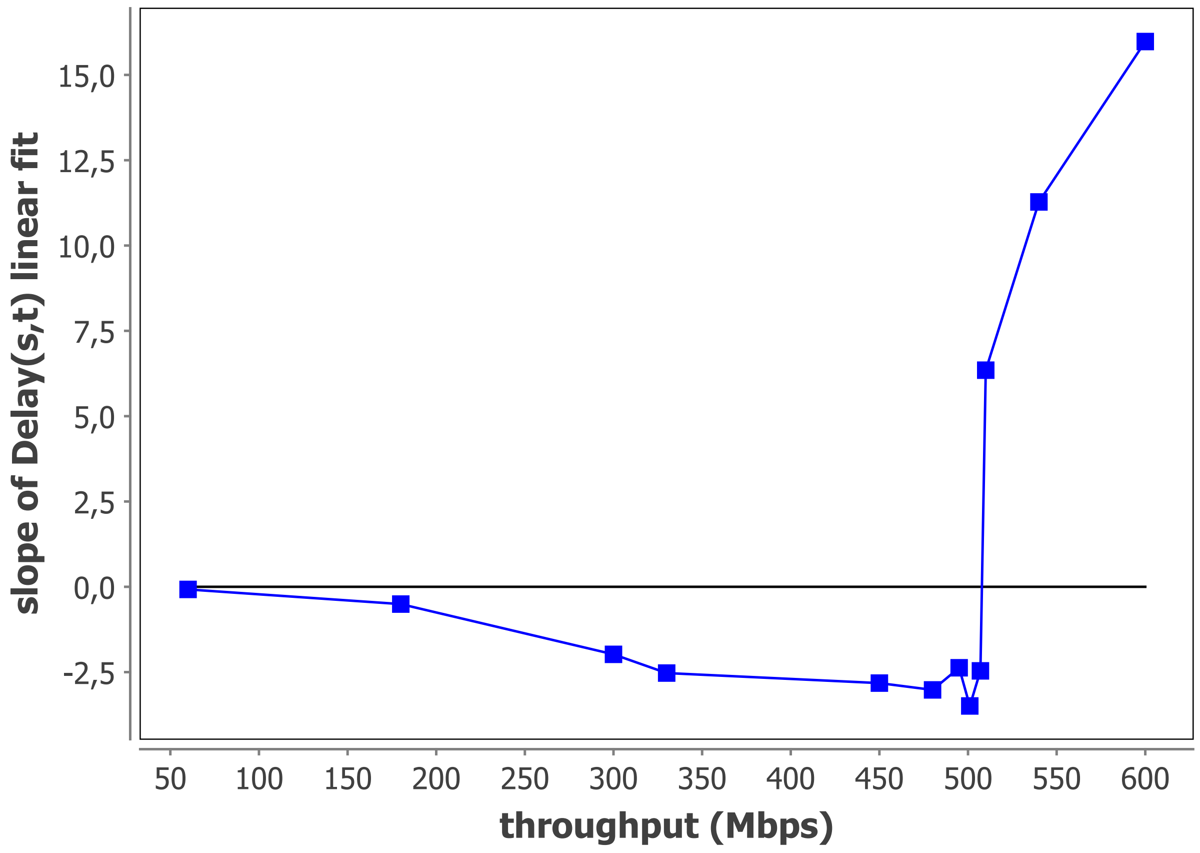

Figure 4.6: Slopes of experiment 2

Throughput capacity is approximately 508 Mbps.

Table 4.3: Incoming and outgoing inter-packet-delay per simulation of experiment 2

s	Input (Mbps)	Input Average Inter-packet delay (media-bus cycles)	StdDev	Output Average Inter-packet delay (media-bus cycles)	StdDev
7	60	2747.71	0.455071	2747.01	113.559
8	180	915.777	0.416712	915.599	145.635
9	300	549.452	0.497989	541.927	119.405
10	330	499.509	0.500199	487.382	148.353
11	450	366.302	0.459476	357.191	122.251
12	480	343.393	0.488761	332.606	165.961
13	495	332.997	0.0576708	325.114	75.7650
14	501	329.013	0.114761	318.797	160.637
15	507	325.107	0.308861	316.513	162.167
16	510	323.197	0.397699	340.587	152.215
17	540	305.243	0.429333	338.316	150.626
18	600	274.713	0.452456	324.726	160.500

Table 4.4: Incoming and outgoing inter-packet-delay per simulation of experiment 3

s	Input (Mbps)	Input Average Inter-packet delay (media-bus cycles)	StdDev	Output Average Inter-packet delay (media-bus cycles)	StdDev
19	180	1026.78	0.415519	1029.92	242.030
20	255	646.423	0.494362	648.530	440.270
21	270	610.488	0.500129	602.042	429.324
22	276	597.231	0.421778	601.459	367.205
23	285	578.390	0.488021	571.173	256.472
24	291	566.424	0.494533	561.737	260.637
25	294	560.652	0.476530	556.111	325.264
26	297	554.997	0.0576708	550.822	150.288
27	300	549.452	0.497989	630.566	254.958
28	303	544.017	0.128090	623.421	325.256

4.6.2 Analysis

The router seem to be at its limit around 297 Mbps, see figure 4.9. This is a severe decrease in throughput capacity compared with a simple routing table complexity of 2, indicating that the routing algorithm implementation have an expensive “inner loop” relative to the constant time cost.

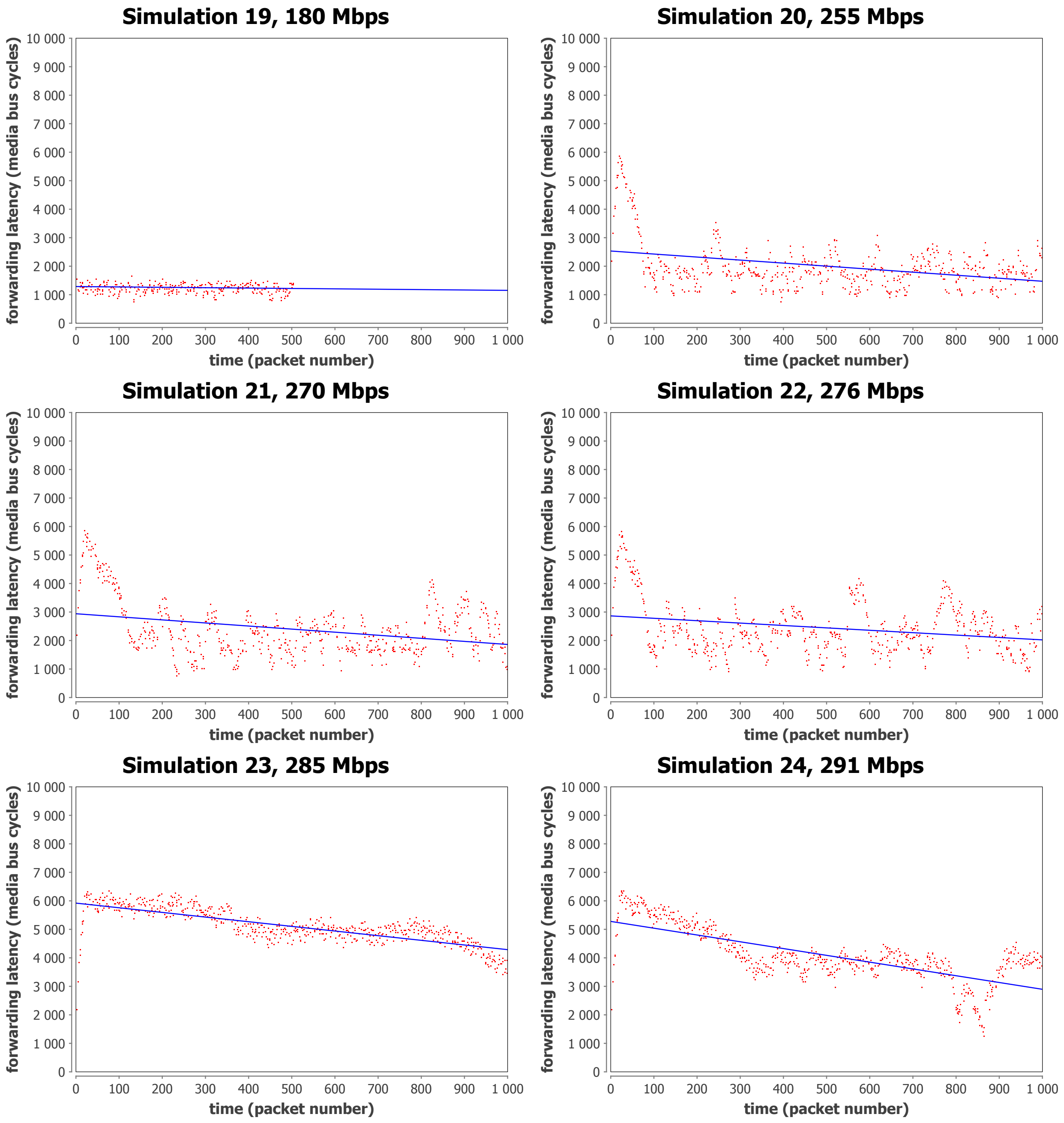

Figure 4.7: Simulations for experiment 3 part 1

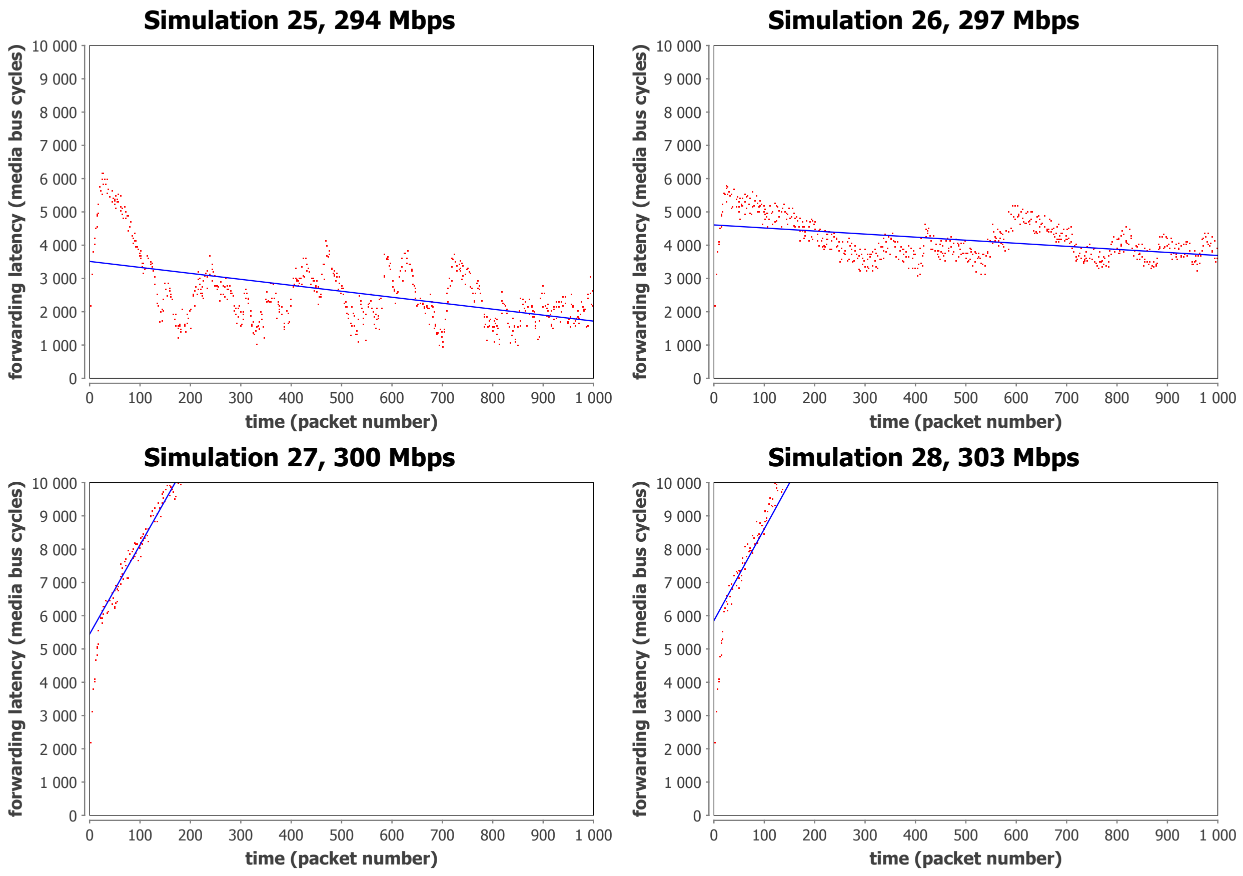

Figure 4.8: Simulations for experiment 3 part 2

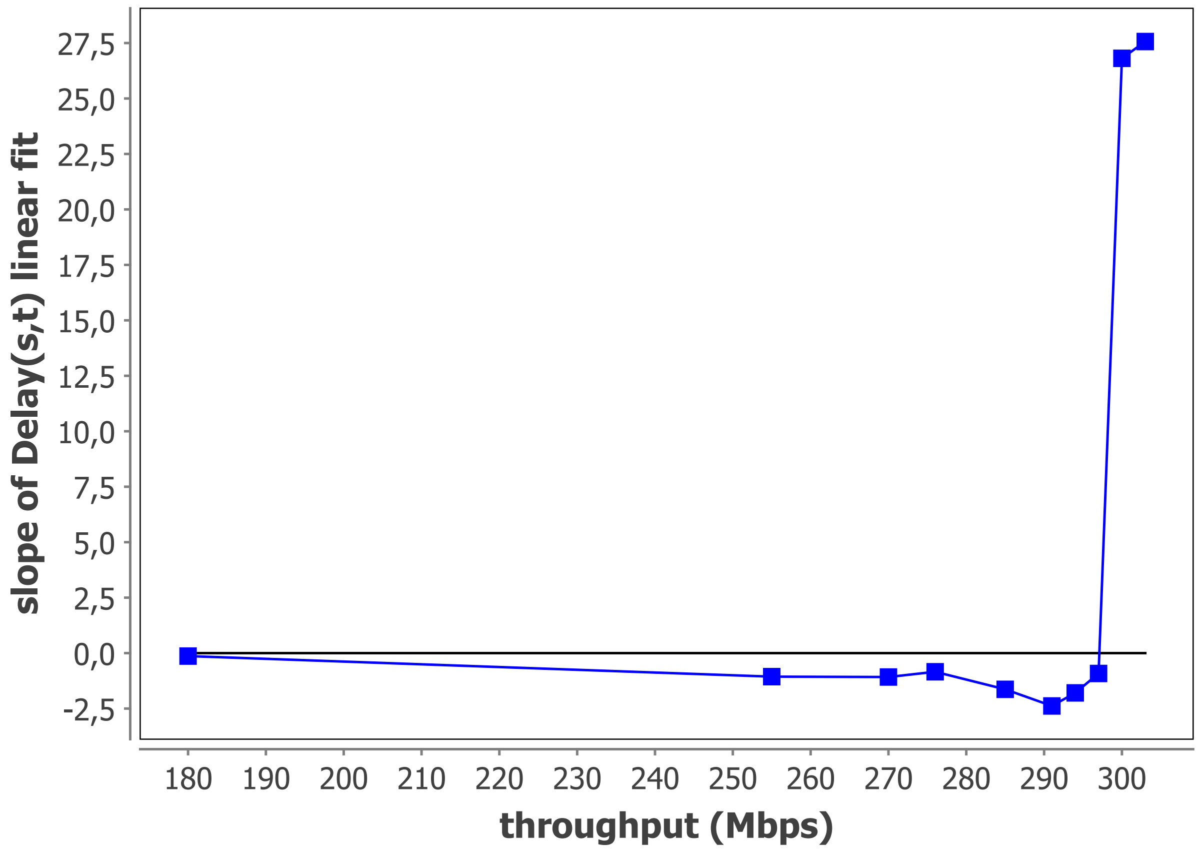

Figure 4.9: Slopes of experiment 3

Throughput capacity is approximately 297 Mbps.

4.7 Experiment 4

4.7.1 Setup

In order to simulate a routing table runtime complexity above 5, the code was slightly changed in this experiment so that the inner loop of the algorithm is run twice rather than once per port per packet. This adds 5 microengine clock cycles to the constant time and doubles the cost of using the routing table. The average routing table runtime complexity is in this experiment 14.16.

4.7.2 Analysis

Jitter is getting noticeably higher in this experiment, see table 4.5, this is also expected due to the variance in inner loop iterations of the algorithm being higher.

4.8 Experiment 5

4.8.1 Setup

In order to simulate a routing table runtime complexity above 5, the code was slightly changed in this experiment so that the inner loop of the algorithm is run three times rather than once per port per packet. This adds 10 microengine clock cycles to the constant time and triples the cost of using the routing table. The routing table runtime complexity is 21.24 for this experiment.

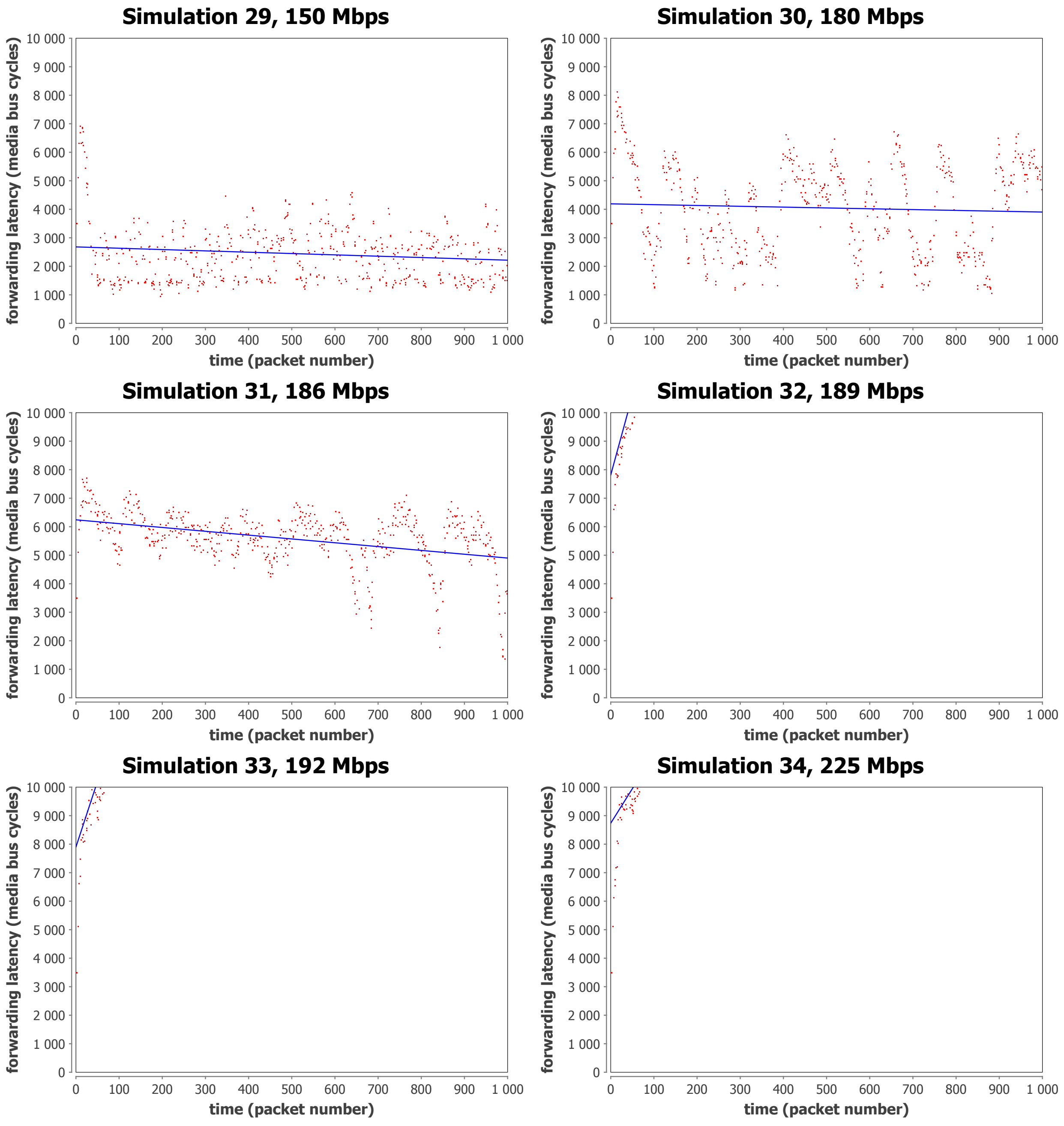

Figure 4.10: Simulations for experiment 4

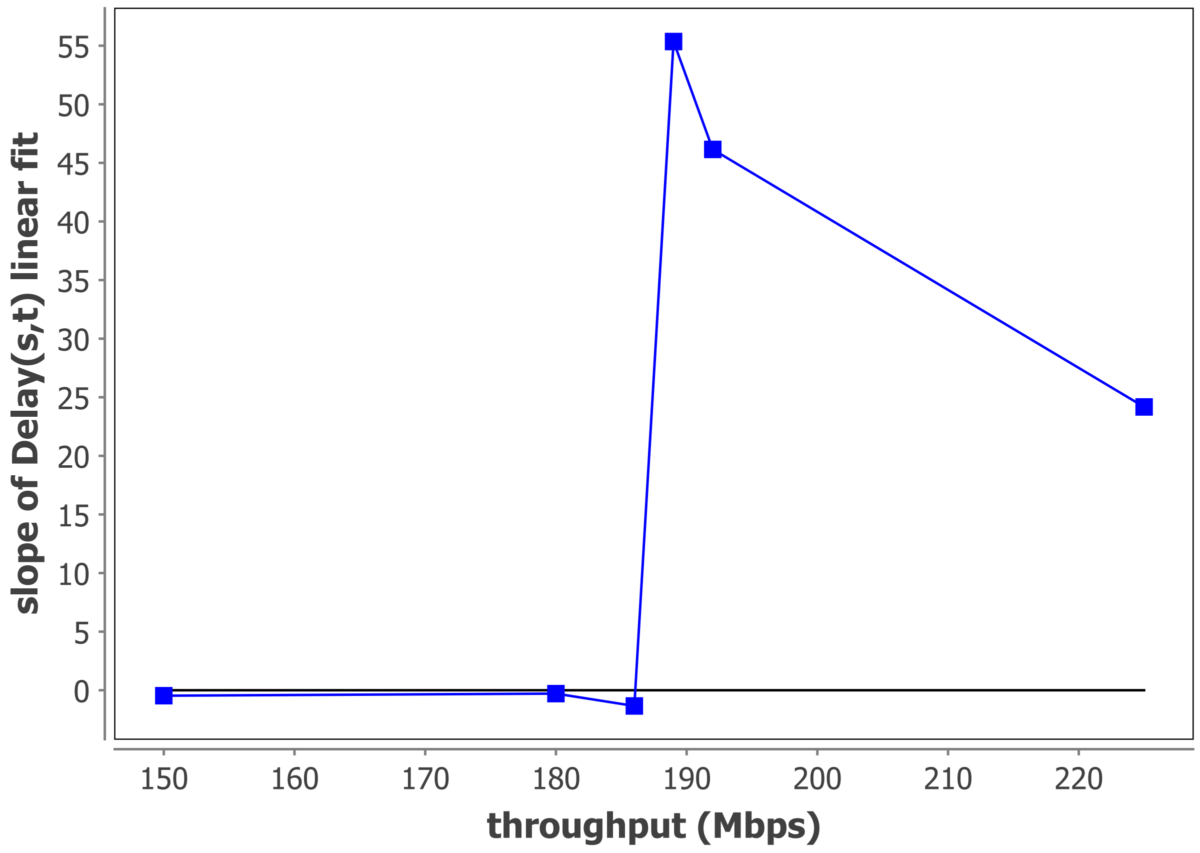

Figure 4.11: Slopes of experiment 4

Throughput capacity is approximately 186 Mbps.

Table 4.5: Incoming and outgoing inter-packet-delay per simulation of experiment 4

s	Input (Mbps)	Input Average Inter-packet delay (media-bus cycles)	StdDev	Output Average Inter-packet delay (media-bus cycles)	StdDev
29	150	1098.90	0.297179	1098.85	888.507
30	180	915.777	0.416712	927.063	664.934
31	186	886.263	0.440687	881.416	561.297
32	189	872.209	0.406741	1023.90	486.746
33	192	858.482	0.499962	996.034	432.342
34	225	732.602	0.489711	802.936	352.152

4.8.2 Analysis

Fewer simulations were done in this experiment, but the results are still clear. The throughput capacity for experiment 5 is about 168 Mbps.

4.9 Summary

4.9.1 Throughput

The algorithm implementation is not optimized and suffer accordingly. With a simple routing table it can keep up with an input rate of almost 510 Mbps which translates into 1020 Mbps outgoing rate when every packet is forwarded to every other port. As the routing table

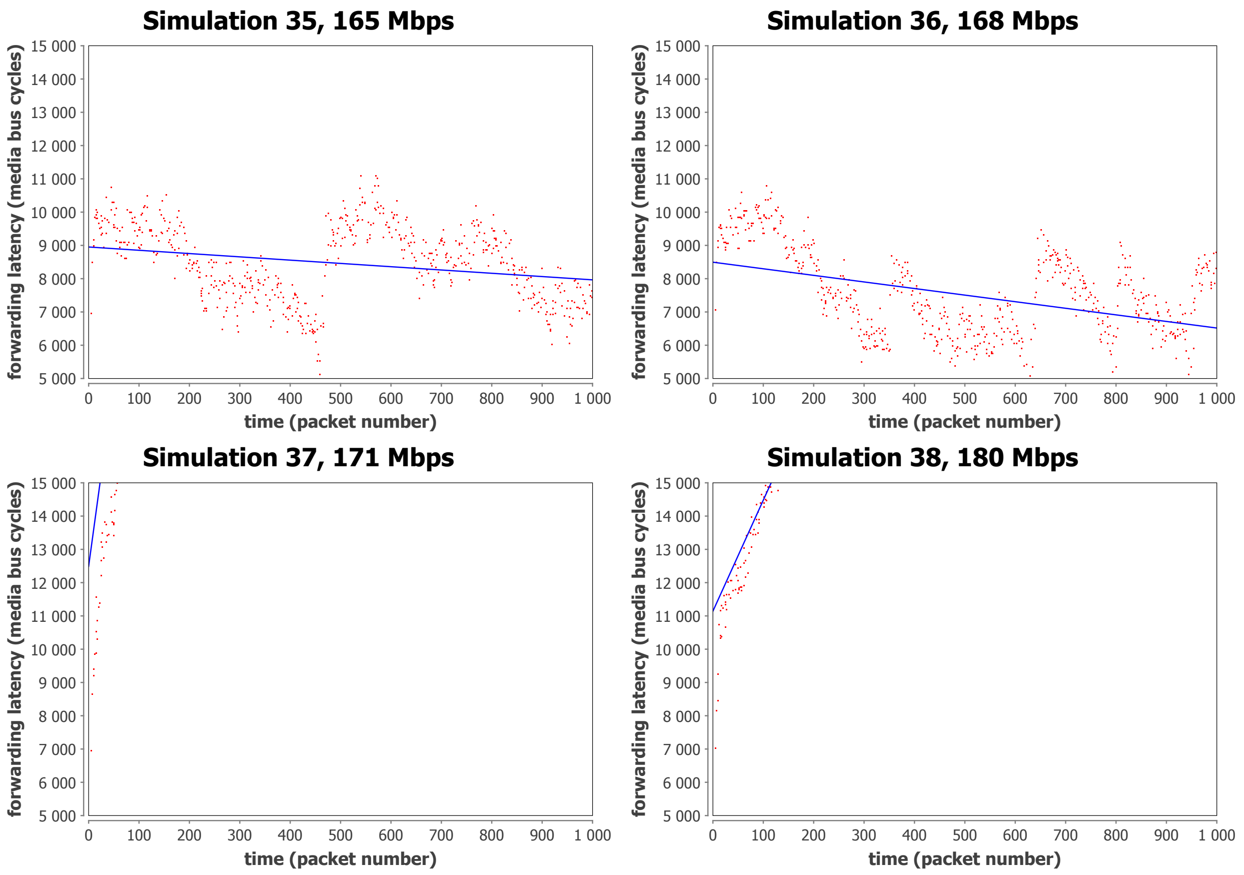

Figure 4.12: Simulations for experiment 5

The Y-axis ranges from 5000 to 15000 in these graphs to better show the results, this differ from all the other experiments that show these graphs with Y-axis from 0 to 10000.

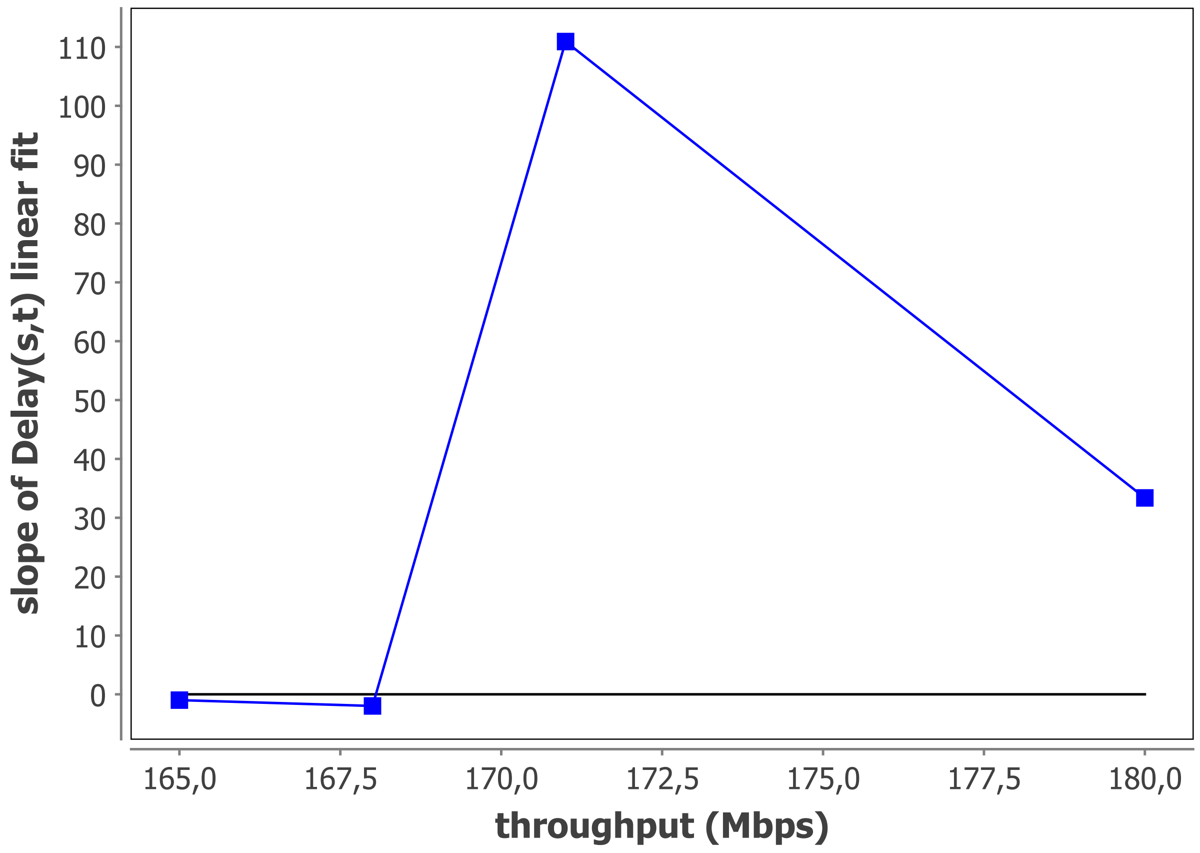

Figure 4.13: Slopes of experiment 5

Throughput capacity is approximately 168 Mbps.

Table 4.6: Incoming and outgoing inter-packet-delay per simulation of experiment 5

s	Input (Mbps)	Input Average Inter-packet delay (media-bus cycles)	StdDev	Output Average Inter-packet delay (media-bus cycles)	StdDev
35	165	999.090	0.286341	993.453	370.152
36	168	981.122	0.327724	977.416	398.056
37	171	963.927	0.260828	1296.24	658.474
38	180	915.777	0.416712	1012.33	577.386

runtime complexity increases, so does the number of filtering operation checks which include a software division, and the throughput capabilities of the router decreases accordingly. With a upper-bound routing table runtime complexity it's possible given enough hardware and a well designed and optimized implementation of the routing algorithm to reach whatever throughput desirable. Even though one can only get so far with code optimization, throughput capacity may be increased through more parallelism usually at the cost of a little higher packet processing times. In figure 4.14 the observed throughput capacity and its fitted function is well below the ideal line. This is implies that our three assumptions of an ideal model were far from holding in these experiments. However, the observed graph has the same general shape, and it's a matter of code design and optimization to bring both the ideal and the observed lines up to production quality which is expected to keep up with line rates of the hardware the router runs.

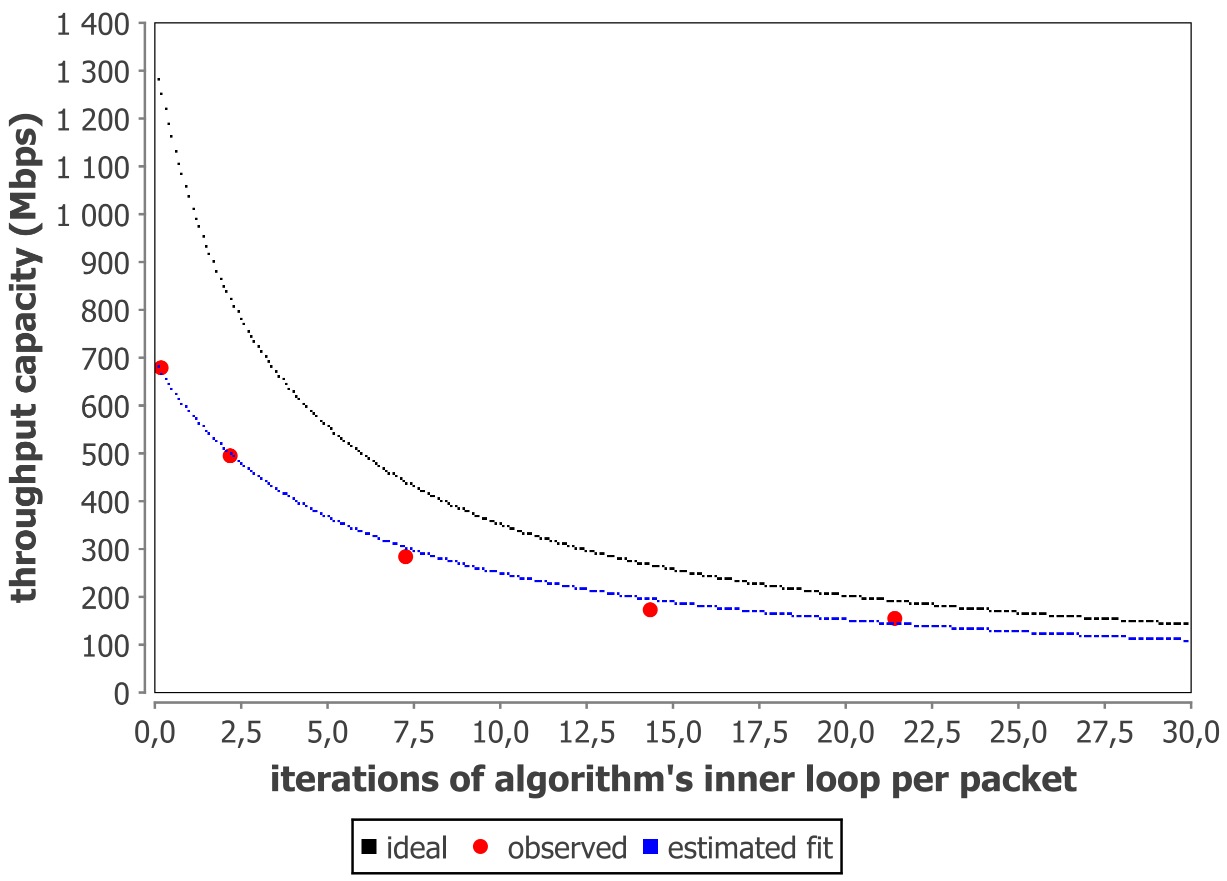

Figure 4.14: Throughput capacity as a function of inner loop iterations

4.9.2 Latency

Theoretically the latency added by the routing microblock is upper bound by 8 times the number of ME cycles along the routing microblock code's longest path assuming that we are able to hide all IO by computing 8 packets in each ME concurrently. It has been shown through simulation in the experiments that the expected latencies depend on how full of packets the microengine pipeline is, though latency is not proportionally higher the fuller the pipeline since a full pipeline will hide a lot more internal IO and a pipeline with a single packet won't hide any of the internal IO related to processing it. It's also important to realize that the longest path in the routing microblock can be very long given a complex routing table. Ideally, the longest path should be static and predictable and not dependent on such runtime variables. It is in other words very hard to predict packet latencies with the current algorithm. These experiments show latencies between 600 and 5000 media bus cycles for stable routers. This translates to approximately 6 and 50 microseconds as the media bus clock runs at 104MHz. In a more optimal implementation, this number should be improved significantly even on such old network processors as IXP2400. It would not be surprising if the improvements yielded merely half the delay on average and worst case within throughput capacity.

4.9.3 Jitter

The tables showing inter packet delay of incoming and outgoing packets reveal that the router does indeed impose a lot of jitter on the fixed rate incoming stream. However, such jitter is easily mitigated through a leaky-bucket algorithm applied after packet routing decisions are made. This would enable more stable outgoing inter packet delays at the cost of a small added total packet processing delay.

4.9.4 Algorithm throughput scalability

The algorithm implementation does suffer from not being production optimized but still exhibit the expected scalability properties. It's worth noting that increasing the number of outgoing ports of the system will raise the requirements of the hardware accordingly since potentially as many more packets are going out of the system as there are ports. It's clear from figure 4.14 that the algorithm does not scale well at all with regards to the runtime dependencies. I.e. the number of iterations of the inner loop of the algorithm required on average per incoming packet.

4.9.5 Conclusions

In order to support a large amount of status variables (flows) and their subscribers we can't use an algorithm that scales on the order of $O(mn)$, but need an $O(m)$ algorithm, where m is the number of links and n the number of subscribers. The point here being that m is fixed and the hardware can be tuned/bought to fit the need of m's size. n however is a dynamic and potentially very large number that if allowed to dictate the performance of the routing algorithm will severely decrease the algorithms scaling potential. However, it's possible to apply a similar logic to n as was done to m. By recognizing that different subscribers may share at most N_{max} number of non-compatible subscriber rates and buying hardware to fit N_{max} and m together. Ideally n should be upper bound by a relatively small number so that the router may exhibit predictable packet processing times and throughput capacities regards of runtime conditions like routing table contents.

An attempt was made to improve the performance by running the algorithm on additional microengines. However, for unknown reasons it did not yield the expected speedups. Going from running the `Route` microblock on 1 microengine to 2 microengines gave speedup close to expectations while adding a third or fourth microengine yielded almost no speedup at all. Why remains an open question, and is possibly due to implementation weaknesses in the prototype.

Overall, the performance of the network processor is much better than that of comparable computer based implementations. Figure 4.15 shows how experiment 2 compares with the computer based C-implementation of a GridStat status router as demonstrated in [19]. Note that both the throughput and the delay axis are using 10-base logarithmic scales, which implies that both throughput and latency performance of the network processor implementation are 1-2 orders of magnitude better than that of a efficient C implementation which was designed to achieve high performance.

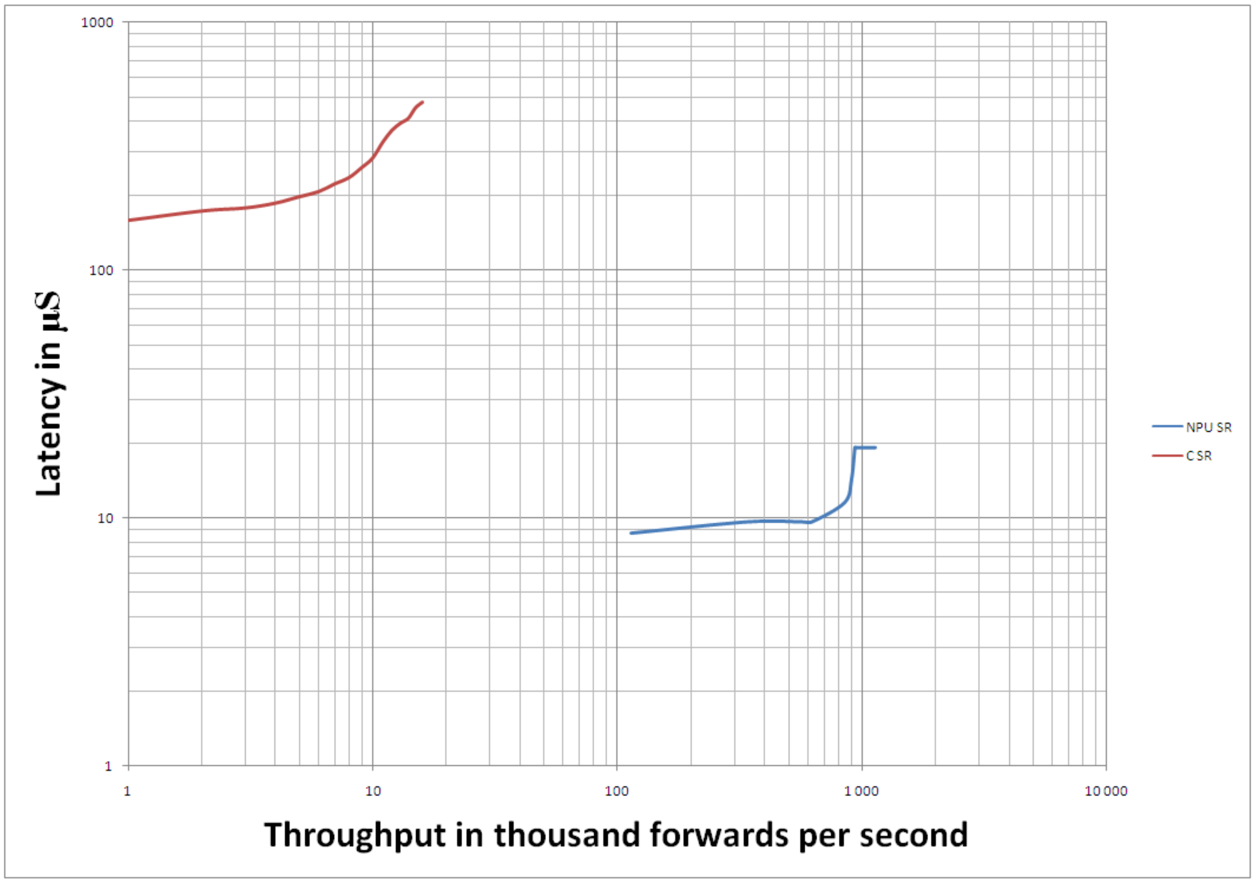

Figure 4.15: Comparing status router performance

Chapter 5

Conclusions

In this thesis we have explained in detail and demonstrated a prototype of a GridStat status router implemented on the IXP2400 network processors. This prototype revealed some challenges of network processor programming in general, like the exposure of complex and parallelized hardware and the need for well optimized code in order to keep the performance of code on par with the expected networking line-speeds. In particular, the GridStat rate-filtering algorithm does not inherently provide packet processing time guarantees, and the potential adverse impact of this has been demonstrated in chapter 4. A solution to provide routing latency guarantees by constraining subscriber options is suggested in the following section which meets the one part of GridStat routing that can not be solved with regards to worst-case merely by upgrading hardware. It is also clear after observing a non-optimal performing prototype, that network processors represent packet processing power that far exceeds that of a regular computer based system of comparable vintage.

5.1 Performance and constraints

GridStat leaf status routers can be implemented on third-generation network processor architectures, and perform well given some constraints. Consider a status router R with M physical links each capable of K Mbps and with a routing table T.

Some scalability challenges of GridStat routing that are also challenges of multicast routing in general can be met by upgrading the hardware of R.

- If T dictate that all incoming packets should be forwarded to every other link ($M - 1$ total), the maximum incoming throughput the router could tolerate without dropping packets is K Mbps regardless of what M is. Therefore, K (and probably also the computational capacity) must be increased in order to increase the incoming throughput capacity of R for all possible cases of T, this requires a hardware upgrade of R.

- M corresponds to the number of iterations in the outer loop of the original routing algorithm. This implies that a larger M requires R to have more computational capacity in order to maintain the same throughput capacity. M is not a free variable, but dictated by the topology of the corresponding GridStat pub-sub routing architecture which in turn is set by design. The GridStat pub-sub topology can therefore be designed in such a way that M is sufficiently small enough for R's computational capacity, and R can be hardware upgraded to match the requirements of a larger but sufficiently small M.

The routing algorithm has one chief scalability challenge. The innermost loop requires an integer modulo operation (which is at least as expensive as a division) where the dividend and divisor are both free variables within the positive integer domain. For each outgoing link in M this operation must in the worst case be executed as many times as there are elements in the corresponding collapsed subscriber list. The number of elements in any collapsed subscriber list is, given sufficiently many subscribers, potentially as many as there are primes in 2^{32}. Even though any collapsed subscriber list size is finite, it's still too large in a bad case that a status router implementation could have its throughput capacity and/or packet processing time suffer unacceptably compared with a trivial subscriber list size. More formally, let v_1 be a status variable and let n be the size of a collapsed subscriber list of v_1. The computational complexity of reaching a forwarding decision for a packet describing only a single sample of v_1 is $O(Mn)$, which can be simplified to $O(n)$ since M is expected to be a sufficiently small fixed value set design time. As seen by simulating the original algorithm on a IXP2400 chip (figure 4.14) increasing n only from 1 to 5 almost halves the status router's throughput capacity due to the high cost of software division. Network processors usually have a computational capacity only a little higher than sufficient to process packets at the hardware's configured network line speed. This implies that increasing n only by a little could decrease the throughput capacity of a network processor well below the line speed of a single physical link. Even with an ideal implementation, where division (and multiplication) was relatively cheap, increasing n would still at some point decrease throughput capacity below line speeds.

One solution to the scalability challenge of the routing algorithm is to introduce subscriber rate restrictions at the management level of the GridStat middleware. Let S_1 be a set of positive integers so that any integer i in S_1 can be evenly divided by all integers in S_1 smaller than i. Let v_1 be a status variable. By restricting all v_1 subscriptions to choose subscriber rates from S_1, any collapsed subscriber list of v_1 would have a size of 0 or 1. If a single set like S_1 is insufficient to meet subscriber rate demands, more such sets could be defined. Let S_2 be a set of positive integers so that any integer i in S_2 can be evenly divided by all integers in S_2 smaller than i. Allowing subscribers of v_1 to choose subscriber rates from $S_1 \cup S_2$ would make any collapsed subscriber list of v_1 have a size of 0, 1, or 2. And so it's possible to define x such sets like S_1 and S_2. By allowing subscribers of v_1 to choose subscribers rates from the union of all x sets, any collapsed subscriber list of v_1 would have a size no larger than x. As long as x is kept small enough, the computational complexity of routing a sample of v_1 is guaranteed $O(1)$ with a constant time that depends on implementation and the size of x which should be chosen at GridStat management level.

A more dynamic approach to solve the scalability challenge of the original routing algorithm could let the GridStat management plane use the knowledge of all the currently registered publishers and subscribers to present new subscribers with restrictions in subscriber rates such that it would not incur routing complexity above a certain threshold in any status router.

In addition to restricting subscribers' possibilities to subscribe at any rates they want, it might be possible to predict the remainders (see line 8 in section 2.1.1) in the routing algorithm. This because the router know the publish intervals and subscriber rates, and should be able to predict the next remainder of any new incoming packet of a stream where the previous packet has already been processed. This could lead to substantial reduction of the inner loop constant time cost and allow far more iterations of it for a given throughput capacity.

All in all even older network processors like the IXP2400 may be expected to do a lot better than current regular computer based systems. In this thesis, with a far from optimal implementation of the routing algorithm, the packet processing times through the status router is between 5 and 50 microseconds and expected to be maybe as little as half that with an optimized implementation. This is better than a relatively fast C implementation [19], and has the benefit of higher throughput because network processor based software can be upscaled well in throughput capacity through increased parallelization.

These are some aspects of network processor hardware and software architecture that might affect the throughput and/or latency performance of a GridStat status router:

- The design and implementation optimization of algorithm to process packets. This is clear after demonstrating a prototype that only utilizes less than 1/3 of the hardware's potential. This requires expert developer knowledge of both the network processor hardware and the accompanying software development kit. As with general programming, the developer of code for a network processor have many options and ways to design and implement the same algorithm, many of which result in performance far below satisfactory when considering expectations of the hardware.

- Average and max code path length taken through the routing algorithm implementation. As this length increases, latency will increase and throughput capacity will decrease. The important aspect here is to be as small as possible, and if more instructions are required, then more microengines must be used to run the code in parallel in order to maintain throughput capacity at acceptable rates.

- Number of microengines N running the routing algorithm in parallel. as N increases, we may generally expect throughput capacity to be proportionally higher, at the cost of a likely negligible latency hit.
- As parallelism increases, the shared buses for access to memory and other subsystems could become congested and have a strong adverse effect on both latency and throughput capacity.
- The instruction set of microengines can potentially make a huge difference. In the case of GridStat routing, hardware support for modulo, or even remainder using reciprocal division could potentially bring the instruction count down by a significant amount.
- Clock frequency of microengines. In general, performance is proportional with clock frequencies. However, unless all subsystems (e.g. memory, and SHaC) also have proportionally higher clock frequencies, it's possible that faster microengines congest these subsystems more and won't get the expected speedup.

5.2 Future work

There are many possibilities to continue research of utilizing network processors in the GridStat middleware data delivery plane, some ideas are:

- Implementing a status router on a “modern” network processor, there are already network processor hardware available that can do more than ten times the throughput of the hardware simulated in this thesis, with at least 40 microengines potentially allowing massively parallelized designs.

- Implement a full-feature status router, including integration with the management layer of GridStat and ensure performance can be upheld while doing lots of management control at runtime.

- Investigate possibilities for line-speed encryption

- Given powerful enough network processors, one could even think of running security applications like intrusion detection in parallel with the normal packet processing pipeline and implementing the appropriate management actions to be taken in GridStat when an intrusion is detected.

References

[1] July 2005. Intel IXP2400 Network Processor Hardware Reference Manual.

[2] Stian F. Abelsen, Harald Gjermundrød, David E. Bakken, and Carl H. Hauser. Adaptive data stream mechanism for control and monitoring applications. In *Proceedings of 1st International Conference on Adaptive and Self-adaptive Systems and Applications (ADAPTIVE'09)*, pages 86–91, Athens, Greece, November 2009.

[3] Yeim-Kuan Chang and Fang-Chen Kuo. Packet processing with blocking for bursty traffic on multi-thread network processor. In *High Performance Switching and Routing, 2009. HPSR 2009. International Conference on*, pages 1–6, June 2009.

[4] Douglas E. Comer. *Network Systems Design using Network Processors.* Pearson Prentice Hall, 2004.

[5] C. Hauser I. Dionysiou H. Gjermundrd L Xu D Bakken, A. Bose and S. Bhowmik. Towards more extensible and resilient real-time information dissemination for the electric power grid. In *Proceedings of Power Systems and Communications Systems for the Future*, September 2002.

[6] Harald Gjermundrød, David E. Bakken, and Carl H. Hauser. Integrating an event pattern mechanism in a status dissemination middleware. In *Proceedings of 1st International Conferences on Pervasive Patterns and Applications (PATTERN'09)*, pages 259–264, Athens, Greece, November 2009.

[7] Harald Gjermundrød, David E. Bakken, Carl H. Hauser, and Anjan Bose. GridStat: A flexible QoS-managed data dissemination framework for the power grid. *IEEE Transactions on Power Delivery*, 24(1):136–143, January 2009.

[8] S. Govind, R. Govindarajan, and J. Kuri. Packet reordering in network processors. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–10, March 2007.

[9] Xiaofeng Guo, Jinquan Dai, Long Li, Zhiyuan Lv, and Prashant R. Chandra. Latency hiding through multithreading on a network processor. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 130–131, New York, NY, USA, 2007. ACM.

[10] Carl H. Hauser, David E. Bakken, Ioanna Dionysiou, K. Harald Gjermundrød, Venkata Irava, Joel Helkey, and Anjan Bose. Security, trust and qos in next-generation control and communication for large power systems. *International Journal of Critical Infrastructures (Interscience)*, 4(1-2):3–16, 2008.

[11] Xianghui Hu, Xinan Tang, and Bei Hua. High-performance ipv6 forwarding algorithm for multi-core and multithreaded network processor. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 168–177, New York, NY, USA, 2006. ACM.

[12] Xin Huang and Tilman Wolf. A methodology for evaluating runtime support in network processors. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 113–122, New York, NY, USA, 2006. ACM.

[13] H. Gjermundrød I. Dionysiou and D. Bakken. Fault tolerance issues in publish-subscribe status dissemination middleware for the electric power grid. In *Supplement of the International Conference on Dependable Systems and Networks (DSN-2002)*, pages B–62–63, June 2002.

[14] Ryan Johnson. Obtaining high performance phasor measurements in a geographically distributed status dissemination network. Master's thesis, Washington State University, August 2005.

[15] Carl Hauser Dave Bakken K. Harald Gjermundrød, Ioanna Dionysiou and Anjan Bose. Flexible and robust status dissemination middleware for the electronic power grid. September 2003.

[16] Duo Liu, Zheng Chen, Bei Hua, Nenghai Yu, and Xinan Tang. High-performance packet classification algorithm for multithreaded ixp network processor. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–25, 2008.

[17] Arindam Mallik, Yu Zhang, and Gokhan Memik. Automated task distribution in multicore network processors using statistical analysis. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 67–76, New York, NY, USA, 2007. ACM.

[18] Gokhan Memik and William H. Mangione-Smith. Evaluating network processors using netbench. *ACM Trans. Embed. Comput. Syst.*, 5(2):453–471, 2006.

[19] Sunil Muthuswamy. System implementation of a real-time, content based application router for a managed publish-subscribe system. Master's thesis, Washington State University, August 2008.

[20] Chris Ostler, Karam S. Chatha, Vijay Ramamurthi, and Krishnan Srinivasan. Ilp and heuristic techniques for system-level design on network processor architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(4):48, 2007.

[21] Yaxuan Qi, Bo Xu, Fei He, Baohua Yang, Jianming Yu, and Jun Li. Towards high-performance flow-level packet processing on multi-core network processors. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 17–26, New York, NY, USA, 2007. ACM.

[22] Erlend S. Viddal, David E. Bakken, Harald Gjermundrød, and Carl H. Hauser. Wide-area actuator rpc over gridstat with timeliness, redundancy, and safety. In *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS'2010*, page Accepted for Publication, Krakow, Poland, February 15-18 2010.

[23] Jean Walrand and Pravin Varaiya. *High-Performance Communication Networks*. Morgan Kaufmann Publishers, 2000.

[24] Ning Weng and Tilman Wolf. Analytic modeling of network processors for parallel workload mapping. *ACM Trans. Embed. Comput. Syst.*, 8(3):1–29, 2009.

[25] T. Wolf and Ning Weng. Runtime support for multicore packet processing systems. *Network, IEEE*, 21(4):29–37, July-August 2007.

[26] Chunqing Wu, Xiangquan Shi, Xuejun Yang, and Jinshu Su. The impact of parallel and multithread mechanism on network processor performance. In *Grid and Cooperative Computing, 2006. GCC 2006. Fifth International Conference*, pages 236–240, Oct. 2006.

[27] Bo Xu, Yaxuan Qi, Fei He, Zongwei Zhou, Yibo Xue, and Jun Li. Fast path session creation on network processors. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on*, pages 573–580, June 2008.

Appendix A

Code Listing

A.1 PacketStreamGenerator.java

Listing A.1: PacketStreamGenerator.java

```java
package thesis;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.zip.CRC32;

import org.apache.commons.codec.DecoderException;
import org.apache.commons.codec.binary.Hex;

public class PacketStreamGenerator {

  /**
   * This program produce data streams that can be used by the ↩
       Development Workbench for IXP network processors by
   * Intel. A single stream for 1 status variable is created. ↩
       The publish interval is hardcoded to 10.
   *
   * Usage: java thesis.PacketStreamGenerator <name of ↩
       datastream> <status variable id> <first timestamp in ↩
       stream>
   *
   */
  public static void main(String[] args) {
    String dataStreamName = args[0];
    int variableId = Integer.parseInt(args[1]);
    int nPackets = Integer.parseInt(args[2]);
    long firstTimestamp = Long.parseLong(args[3]);
    int pubIntMillis = 10; // hard-coded for now

    CRC32 crc32 = new CRC32();

    File f = new File(dataStreamName + ".strm");
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream(f);
      PrintStream out = new PrintStream(fos);

      // generate stream header
      out.println("# IXP1200 Developer Workbench Data Stream ↩
         File");
      out.println("# Format Version 6.00");
      out.println("#********  Do not edit this file **********"↩
         );
      out.println("");
      out.println("# Begin Data Stream " + dataStreamName);
      out.println("STREAM_TYPE = Ethernet IP");

      long timestamp = firstTimestamp;

      for (int i = 0; i < nPackets; ++i) {

        // Define packet data
        String etherHeaderFixedData = "3↩
           a3a7b7b7bcc4646464646460800";
        String ipHeaderFixedData = "45000030000000004063500↩
           c0a80032c0a80046";
        ByteArrayOutputStream baos = new ByteArrayOutputStream()↩
           ;
        PrintStream ps = new PrintStream(baos);
        ps.format("%016x%08x%024x%08x", timestamp, variableId, (↩
           int) 0, i + 1); // print data payload in hex
        String payloadFixedData = new String(baos.toByteArray())↩
           ;

        // compute packet crc32
        byte[] binaryPacketData;
        binaryPacketData = Hex.decodeHex((etherHeaderFixedData +
            ipHeaderFixedData + payloadFixedData)
            .toCharArray());
        crc32.reset();
        crc32.update(binaryPacketData);
        baos.reset();
        ps.format("%08x", crc32.getValue());
        String etherTrailerFixedDataWrongByteOrder = new String(
           baos.toByteArray());
        String etherTrailerFixedData =
           etherTrailerFixedDataWrongByteOrder.substring(6, 8)
            + etherTrailerFixedDataWrongByteOrder.substring(4,
               6)
            + etherTrailerFixedDataWrongByteOrder.substring(2,
               4)
            + etherTrailerFixedDataWrongByteOrder.substring(0,
               2);

        // generate frame in file stream
        out.println("# Begin Ethernet Frame");
        out.println("# Begin Ethernet Header");
        out.println("FIXED_DATA = " + etherHeaderFixedData);
        out.println("# End Ethernet Header");
        out.println("# Begin Ip Packet");
        out.println("# Begin Ip Header");
        out.println("COMPUTE_CHECKSUM = TRUE");
        out.println("COMPUTE_PACKET_LENGTH = TRUE");
        out.println("FIXED_DATA = " + ipHeaderFixedData);
        out.println("# End Ip Header");
        out.println("# Begin Data Payload");
        out.println("USE_FILL_PATTERN = FALSE");
        out.println("FILL_PATTERN_TYPE = 15");
        out.println("FILL_START_VALUE = " + (i * 14));
        out.println("DATA_PAYLOAD_SIZE = 28");
        out.println("DATA_PAYLOAD = " + payloadFixedData);
        out.println("# End Data Payload");
        out.println("# End Ip Packet");
        out.println("# Begin Ethernet Trailer");
        out.println("FIXED_DATA = " + etherTrailerFixedData);
        out.println("# End Ethernet Trailer");
        out.println("# End Ethernet Frame");

        timestamp += pubIntMillis;
      }

      out.println("# End Data Stream");
      out.println();

    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (DecoderException e) {
      e.printStackTrace();
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }

}
```

A.2 RTSetupCodeGenerator.java

Listing A.2: RTSetupCodeGenerator.java

```
package thesis;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

public class RTSetupCodeGenerator {

  static class ReciprocalResult {
    int divisor;
    int nbits;
    long reciprocal;
  }

  /*
   *
   * Input: nbits number of significant bits used in reciprocal.↩
      divisor divisor.
```

```
 *
 * Output: returns reciprocal with nbits significant bits.
 */
static void computeReciprocal(int divisor, ReciprocalResult rr) {
  int z;
  long r;
  double f_tmp;
  rr.divisor = divisor;

  // find z so that 0.5 <= (1/d * 2^z) < 1
  // then 1/d will have z leading zeros between the decimal point and its
  // most significant set bit
  z = 0;
  f_tmp = 1.0 / divisor;
  while (0.5 > f_tmp || f_tmp >= 1) {
    z++;
    // f_tmp = 2^z / d
    f_tmp = ((double) (2 << (z - 1))) / divisor;
  }

  // compute (int) ( 2^(z+n+1) / d )
  r = (2l << (z + 32)) / divisor;

  rr.nbits = 32 + z;
  rr.reciprocal = (long) (r >> 1);
}

static void generate_init_rtent_function() {
  System.out.println("");
  System.out.println("int init_rtent( int addr, int varid, int pubint ) {");
  System.out.println("  // VID_ENTRY* next");
  System.out.println("  set_sram(addr, 0x0);");
  System.out.println("");
  System.out.println("  // long variableid");
  System.out.println("  addr = addr + 4;");
  System.out.println("  set_sram(addr, varid);");
  System.out.println("");
  System.out.println("  // long pub_interval, hardcoded as 10 for all status variables");
  System.out.println("  addr = addr + 4;");
  System.out.println("  set_sram(addr, 10);");
  System.out.println("");
```

```
    System.out.println("  // long reserved");
    System.out.println("  addr = addr + 4;");
    System.out.println("  set_sram(addr, 0x0);");
    System.out.println("");
    System.out.println("  // FILTER_ENTRY subint[NO_PORTS][↩
       MAX_SUB_INTS]");
    System.out.println("  addr = addr + 4;");
    System.out.println("");
    System.out.println("  // SUBSCRIBER_ENTRY *sub_head[NO_PORTS↩
       ]");
    System.out.println("  addr = addr + (8*3*6);");
    System.out.println("  set_sram(addr, 0x0);");
    System.out.println("  addr = addr + 4;");
    System.out.println("  set_sram(addr, 0x0);");
    System.out.println("  addr = addr + 4;");
    System.out.println("  set_sram(addr, 0x0);");
    System.out.println("");
    System.out.println("  return 1;");
    System.out.println("}");
    System.out.println("");
  };

  static void generate_rt_init_header() {
    System.out.println("int rt_init() {");
    System.out.println("  int offset;");
    System.out.println("  int addr;");
    System.out.println("");
    System.out.println("  // clear entire routing table");
    System.out.println("  addr = GRIDSTAT_RTABLE_BASE;");
    System.out.println("  offset = addr + GRIDSTAT_RTABLE_SIZE;"↩
       );
    System.out.println("  init_sram(0x0, addr, offset);");
    System.out.println("");
  }

  static void generate_rt_init_body(RoutingTable rt) {
    ReciprocalResult rr = new ReciprocalResult();
    for (StatusVariableSubscriptions sv : rt.↩
       statusVariableSubscriptions) {

      System.out.println("  // VARIABLEID " + sv.variableid);
      System.out.println("");
      System.out.println("  offset = " + sv.variableid + ";");
      System.out.println("  addr = GRIDSTAT_RTABLE_BASE + offset↩
          * GRIDSTAT_RTENT_SIZE;");
      System.out.println("  init_rtent(addr, " + sv.variableid + ", 10);");
      System.out.println("  addr = addr + 16;");
      System.out.println("");

      if (sv.variableid == 0) {
        continue;
      }

      for (int p = 0; p < NO_PORTS; ++p) {
        System.out.println("  // port " + p);
        System.out.println("");
        for (int i = 0; i < MAX_DIVISORS + 1; ++i) {
          if (sv.divisors[p][i] != 0) {
            computeReciprocal(sv.divisors[p][i], rr);
            long rtd = (rr.divisor & 0x07ffffff) | (rr.nbits << 27);
            System.out.println("  // subscriber " + (i + 1) + " with divisor " + sv.divisors[p][i]);
            System.out.println("  set_sram(addr, 0x" + Long.toHexString(rtd) + ");");
            System.out.println("  addr = addr + 4;");
            System.out.println("  set_sram(addr, 0x" + Long.toHexString(rr.reciprocal) + ");");
            System.out.println("  addr = addr + 4;");
            System.out.println("");
          } else {
            System.out.println("  addr = addr + 8;");
          }
        }
        System.out.println("");
        System.out.println("");
      }
    }
  }

  static void generate_rt_init_footer() {
    System.out.println("");
    System.out.println("  return 1;");
    System.out.println("}");
    System.out.println("");
  }

  public static class Subscription {
    int variableid;
    int port;
    int divisor;
  };

  public static final int NO_PORTS = 3;
  public static final int MAX_DIVISORS = 5;

  public static class StatusVariableSubscriptions {
    int variableid;
    int[] n = new int[NO_PORTS]; // number of active ↩
       subscriptions
    int[][] divisors = new int[NO_PORTS][MAX_DIVISORS + 1];
  }

  public static class RoutingTable {
    List<StatusVariableSubscriptions> ↩
       statusVariableSubscriptions = new ArrayList<↩
       StatusVariableSubscriptions>();

    public StatusVariableSubscriptions find(int variableId) {
      for (StatusVariableSubscriptions svs : ↩
         statusVariableSubscriptions) {
        if (svs.variableid == variableId) {
          return svs;
        }
      }
      return null;
    }
  }

  public static RoutingTable createRoutingTable(List<Integer> ↩
     args) {
    // prepare routing table based on user input
    List<Subscription> subscriptions = new ArrayList<↩
       Subscription>();
    for (int i = 0; i < args.size(); i += 3) {
      Subscription s = new Subscription();
      s.variableid = args.get(i);
      s.port = args.get(i + 1);
      s.divisor = args.get(i + 2);
      subscriptions.add(s);
    }
    Collections.sort(subscriptions, new Comparator<Subscription↩
       >() {
      @Override
      public int compare(Subscription arg0, Subscription arg1) {
        return arg0.port - arg1.port;
      }
    });
    Collections.sort(subscriptions, new Comparator<Subscription↩
       >() {
      @Override
      public int compare(Subscription arg0, Subscription arg1) {
        return arg0.variableid - arg1.variableid;
      }
    });
    RoutingTable rt = new RoutingTable();
    StatusVariableSubscriptions sv = new ↩
       StatusVariableSubscriptions();
    rt.statusVariableSubscriptions.add(sv); // variableid 0
    for (Subscription s : subscriptions) {
      if (sv.variableid != s.variableid) {
        sv = new StatusVariableSubscriptions();
        sv.variableid = s.variableid;
        rt.statusVariableSubscriptions.add(sv);
      }
      if (sv.n[s.port] >= MAX_DIVISORS) {
        throw new IndexOutOfBoundsException("too many ↩
           subscriptions for variableid " + s.variableid
           + " and port " + s.port);
      }
      sv.divisors[s.port][sv.n[s.port]++] = s.divisor;
    }
    return rt;
  }

  /**
   * This program will generate a script to be used by the ↩
      Development Workbench to set up the routing table in SRAM.
   * The script is printed to standard out.
   *
   * Usage: java thesis.RTSetupCodeGenerator [<status variable ↩
      id> <port> <subscriber rate >]...
   *
   */
  public static void main(String[] args) {

    // parse arguments
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 0; i < args.length; i += 3) {
      list.add(new Integer(args[i]));
      list.add(new Integer(args[i + 1]));
      list.add(new Integer(args[i + 2]));
    }

    // setup internal routing table
    RoutingTable rt = createRoutingTable(list);

    // generate code
    generate_init_rtent_function();
    generate_rt_init_header();
    generate_rt_init_body(rt);
    generate_rt_init_footer();
  }
}
```

A.3 Packet.java

Listing A.3: Packet.java

```
package thesis;

/**
 * An instance of this class represents one incoming packet and hold performance data of that packet as computed by
 * looking at the original packet logs from the simulator. Each incoming packet has potentially N-1 target ports, and
 * the delay between each incoming packet's EOP and its corresponding outgoing packets' SOP form the basis of packet
 * processing durations that almost all other performance measurements are based on.
 *
 */
public class Packet {

  // number of physical ports connected to network processor
  static final int N = 3;

  static class TransmittedPacket {
    boolean forwarded = false;
    long tx_sop;
    long tx_eop;
    Double avg5Delay = null;
```

```java
  }

  static final String[] CSV_HEADERS = { "Packet", "AVG TX Delay"↩
      , "AVG TX last 5 AVG Delay", "RX Port",
      "RX PacketNum", "Receive SOP", "Receive EOP", "Timestamp",↩
          "Variable ID", "TX0 Forwarded", "TX0 SOP",
      "TX0 EOP", "TX0 Delay", "TX0 last 5 AVG Delay", "TX1 ↩
         Forwarded", "TX1 SOP", "TX1 EOP", "TX1 Delay",
      "TX1 last 5 AVG Delay", "TX2 Forwarded", "TX2 SOP", "TX2 ↩
         EOP", "TX2 Delay", "TX2 last 5 AVG Delay" };

  static final String csvHeader;

  static {
    StringBuilder sb = new StringBuilder();
    sb.append(CSV_HEADERS[0]);
    for (int i = 1; i < CSV_HEADERS.length; ++i) {
      sb.append("\t").append(CSV_HEADERS[i]);
    }
    csvHeader = sb.toString();
  }

  String packetData;

  int packet;
  int variableId;
  long timestamp;

  int rx_packetNum;
  int rx_port;
  long rx_sop;
  long rx_eop;

  Double avgTxDelay = null;
  Double avgLast5AvgTxDelay = null;

  TransmittedPacket[] tp = new TransmittedPacket[N];

  @Override
  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + (int) (timestamp ^ (timestamp >>> ↩
       32));
    result = prime * result + variableId;
    return result;
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Packet other = (Packet) obj;
    if (timestamp != other.timestamp)
      return false;
    if (variableId != other.variableId)
      return false;
    return true;
  }

  String toCsv() {
    StringBuilder sb = new StringBuilder();
    sb.append(packet).append("\t").append(avgTxDelay == null ? "" : avgTxDelay).append("\t").append(
        avgTxDelay == null ? "" : avgLast5AvgTxDelay == null ? "" : avgLast5AvgTxDelay).append("\t").append(
        rx_port).append("\t").append(rx_packetNum).append("\t").append(rx_sop).append("\t").append(rx_eop)
        .append("\t").append(timestamp).append("\t").append(variableId);
    for (int i = 0; i < N; ++i) {
      if (tp[i] != null) {
        TransmittedPacket t = tp[i];
        sb.append("\t").append(t.forwarded ? "1" : "").append("\t").append(t.tx_sop).append("\t").append(
            t.tx_eop).append("\t").append(t.tx_sop - rx_eop).append("\t").append(
            (t.avg5Delay == null) ? "" : t.avg5Delay);
      } else {
        sb.append("\t\t\t\t\t");
      }
    }
    return sb.toString();
  }
}
```

A.4 GraphProducer.java

Listing A.4: GraphProducer.java

```java
package thesis;

import java.awt.Color;
import java.awt.Rectangle;
import java.awt.geom.Ellipse2D;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.io.UnsupportedEncodingException;
import java.io.Writer;
import java.util.ArrayList;
import java.util.List;

import org.apache.batik.dom.GenericDOMImplementation;
import org.apache.batik.svggen.SVGGraphics2D;
import org.apache.batik.svggen.SVGGraphics2DIOException;
import org.apache.commons.math.stat.regression.SimpleRegression;
import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.DatasetRenderingOrder;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.chart.plot.XYPlot;
import org.jfree.chart.renderer.xy.XYDotRenderer;
import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
import org.jfree.data.xy.XYDataItem;
import org.jfree.data.xy.XYSeries;
import org.jfree.data.xy.XYSeriesCollection;
import org.jfree.ui.RectangleInsets;
import org.w3c.dom.DOMImplementation;
import org.w3c.dom.Document;

/**
 * This class is capable of producing JFreeChart graphs and ↩
     rendering these as SVG files. Three different types of
 * graphs are implemented:
 *
 * 1. One graph per simulation representing the packet ↩
     processing duration (delay) as a function of time (packet
 * number). Also in the same plot an estimated linear fit of ↩
    that delay function. One observation per incoming packet in
 * every simulation.
 *
 * 2. One graph per experiment representing the slope parts of ↩
    the estimated linear fits from (1). One observation per
 * simulation in experiment.
 *
 * 3. A single graph depicting ideal, observations of, and ↩
    estimated fit of the throughput capacity function as seen
 * across all experiments and simulations.
 *
 */
public class GraphProducer {

  static JFreeChart createDelayWithLinearFit(List<Packet> ↩
     packets, SimpleRegression sr, String title, int yAxisStart)↩
      {
    // create new chart
    JFreeChart chart = ChartFactory.createScatterPlot(title, "↩
       time (packet number)", "forwarding latency (media bus ↩
       cycles)", null,
        PlotOrientation.VERTICAL, false, true, true);

    // general settings
    chart.setPadding(new RectangleInsets(0, 0, 0, 5));
    XYPlot xyPlot = chart.getXYPlot();
    xyPlot.setBackgroundPaint(null);
    xyPlot.setDatasetRenderingOrder(DatasetRenderingOrder.↩
       FORWARD);

    // X-axis
    xyPlot.getDomainAxis().setAutoRange(false);
    xyPlot.getDomainAxis().setRange(0, 1000);

    // Y-axis
    xyPlot.getRangeAxis().setAutoRange(false);
    xyPlot.getRangeAxis().setRange(yAxisStart, yAxisStart + ↩
       10000);

    // delay dataset
    XYSeries delaySeries = new XYSeries("observations");
    for (Packet p : packets) {
      delaySeries.add(p.packet, p.avgTxDelay);
    }
    xyPlot.setDataset(0, new XYSeriesCollection(delaySeries));

    // delay rendering
    XYDotRenderer dotRenderer = new XYDotRenderer();
    dotRenderer.setDotWidth(1);
    dotRenderer.setDotHeight(1);
    dotRenderer.setSeriesPaint(0, Color.RED);
    xyPlot.setRenderer(0, dotRenderer);

    // regression dataset
    XYSeries linearSeries = new XYSeries("linear fit");
    linearSeries.add(0, sr.predict(0));
    linearSeries.add(1000, sr.predict(1000));
    xyPlot.setDataset(1, new XYSeriesCollection(linearSeries));

    // regression rendering
    XYLineAndShapeRenderer lineRenderer = new ↩
       XYLineAndShapeRenderer();
    lineRenderer.setBaseShapesVisible(false);
    lineRenderer.setSeriesPaint(0, Color.BLUE);
    xyPlot.setRenderer(1, lineRenderer);

    return chart;
  }

  static JFreeChart createSlopeOfDelay(XYSeries slopeSeries) {
    // create new chart
    JFreeChart chart = ChartFactory.createXYLineChart(null, "↩
       throughput (Mbps)", "slope of Delay(s,t) linear fit", ↩
       null,
        PlotOrientation.VERTICAL, false, true, true);

    /*
     * general settings
     */
    XYPlot xyPlot = chart.getXYPlot();
    xyPlot.setBackgroundPaint(null);

    /*
     * slope dataset. One sample per simulation.
     */
    xyPlot.setDataset(0, new XYSeriesCollection(slopeSeries));
    // render in bluw with straight lines between markers.
    XYLineAndShapeRenderer splineRenderer = new ↩
       XYLineAndShapeRenderer();
    splineRenderer.setSeriesPaint(0, Color.BLUE);
    xyPlot.setRenderer(0, splineRenderer);

    /*
     * y=0 series. slope dataset will intersect with this line ↩
        approximately where the throughput capacity is.
     */
    XYSeries linearSeries = new XYSeries("y=0");
    linearSeries.add(slopeSeries.getMinX(), 0);
    linearSeries.add(slopeSeries.getMaxX(), 0);
    xyPlot.setDataset(1, new XYSeriesCollection(linearSeries));
    // render as black solid line with no markers
    XYLineAndShapeRenderer lineRenderer = new ↩
       XYLineAndShapeRenderer();
    lineRenderer.setSeriesShapesVisible(0, false);
    lineRenderer.setSeriesPaint(0, Color.BLACK);
    xyPlot.setRenderer(1, lineRenderer);

    return chart;
  }

  /*
   * The following three functions are based on the equation:
   *
   * throughput capacity = (me * clock * bpp) / (k1 + (k2 * x))
   */

  static double throughputCapacity(int nMe, int clockMHz, int ↩
     bpp, double k1, double k2, double x) {
    return (nMe * clockMHz * bpp) / (k1 + k2 * x);
  }

  static double estimateK1(int nMe, int clockMHz, int bpp, ↩
     double throughput, double k2, double x) {
    return ((nMe * clockMHz * bpp) / throughput) - (k2 * x);
  }

  static double estimateK2(int nMe, int clockMHz, int bpp, ↩
     double throughput, double k1, double x) {
    return (((nMe * clockMHz * bpp) / throughput) - k1) / x;
  }

  @SuppressWarnings("unchecked")
  static JFreeChart createThroughputCapacityFunction(boolean ↩
     drawObserved, int nMe, int clockMHz, int bitsPerPacket,
      int idealK1, int idealK2, XYSeries observations) {

    // create new chart
    JFreeChart chart = ChartFactory.createXYLineChart(null, "↩
       iterations of algorithm's inner loop per packet",
        "throughput capacity (Mbps)", null, PlotOrientation.↩
           VERTICAL, true, true, true);

    /*
     * general settings
     */
    XYPlot xyPlot = chart.getXYPlot();
    xyPlot.setBackgroundPaint(null);
    xyPlot.setDatasetRenderingOrder(DatasetRenderingOrder.↩
       FORWARD);
    // X-axis
    // compute xAxisRange to the closest even 10 greater than ↩
       all x observations.
    int xAxisRange = 10 * (((int) observations.getMaxX() + 10) /↩
        10);
    xyPlot.getDomainAxis().setAutoRange(false);
    xyPlot.getDomainAxis().setRange(0, xAxisRange);
    // Y-axis
    // compute yAxisRange to the closest even 100 greater than y↩
        when x=0.
    int yAxisRange = 100 * (((int) throughputCapacity(nMe, ↩
       clockMHz, bitsPerPacket, idealK1, idealK2, 0) + 100) / ↩
       100);
    xyPlot.getRangeAxis().setAutoRange(false);
    xyPlot.getRangeAxis().setRange(0, yAxisRange);

    /*
     * ideal dataset
     */
    XYSeries idealSeries = new XYSeries("ideal");
    for (double x = 0; x <= xAxisRange; x = x + 0.1) {
      idealSeries.add(x, throughputCapacity(nMe, clockMHz, ↩
         bitsPerPacket, idealK1, idealK2, x));
    }
    xyPlot.setDataset(0, new XYSeriesCollection(idealSeries));
    // ideal rendering, as dots
    XYDotRenderer dotRenderer = new XYDotRenderer();
    dotRenderer.setDotWidth(1);
    dotRenderer.setDotHeight(1);
    dotRenderer.setSeriesPaint(0, Color.BLACK);
    xyPlot.setRenderer(0, dotRenderer);

    /*
     * observed series
     */
    if (drawObserved) {
      xyPlot.setDataset(1, new XYSeriesCollection(observations))↩
          ;
      // render observations as large dots with no connecting ↩
          line
      XYLineAndShapeRenderer simulatedRenderer = new ↩
          XYLineAndShapeRenderer();
      simulatedRenderer.setSeriesLinesVisible(0, false);
      simulatedRenderer.setSeriesPaint(0, Color.RED);
      simulatedRenderer.setSeriesShape(0, new Ellipse2D.Double↩
          (0, 0, 5, 5));
      xyPlot.setRenderer(1, simulatedRenderer);
    }

    /*
     * estimate K1 and K2 based on observations
     */
    double k1 = 0;
    double k2 = 0;
    double k2Sum = 0;
    int k2Count = 0;
    for (XYDataItem item : (List<XYDataItem>) observations.↩
        getItems()) {
      if (item.getXValue() == 0) {
        // k1 is based on x=0 observation
        k1 = estimateK1(nMe, clockMHz, bitsPerPacket, item.↩
            getYValue(), 1 /* K2 have no impact when x is 0 */, ↩
            0);
        break;
      }
    }
    for (XYDataItem item : (List<XYDataItem>) observations.↩
        getItems()) {
      if (item.getXValue() > 0) {
        k2Sum += estimateK2(nMe, clockMHz, bitsPerPacket, item.↩
            getYValue(), k1, item.getXValue());
        k2Count++;
      }
    }
    if (k2Count > 0) {
      // k2 is based on the average estimate of all experiments with x>0
      k2 = k2Sum / k2Count;
    }

    /*
     * fit a line to observations
     */
    if (drawObserved) {
      XYSeries estimatedFitSeries = new XYSeries("estimated fit");
      for (double x = 0; x <= xAxisRange; x = x + 0.1) {
        estimatedFitSeries.add(x, throughputCapacity(nMe, clockMHz, bitsPerPacket, k1, k2, x));
      }
      xyPlot.setDataset(2, new XYSeriesCollection(estimatedFitSeries));
      // fitted line rendered as dots
      XYDotRenderer estimatedFitRenderer = new XYDotRenderer();
      estimatedFitRenderer.setDotWidth(1);
      estimatedFitRenderer.setDotHeight(1);
      estimatedFitRenderer.setSeriesPaint(0, Color.BLUE);
      xyPlot.setRenderer(2, estimatedFitRenderer);
    }

    return chart;
  }

  static final int DELAY_CHART_WIDTH = 500;
  static final int DELAY_CHART_HEIGHT = 350;

  // At most 6 graphs per page
  static Rectangle[] BOUNDS = new Rectangle[6];

  static {
    BOUNDS[0] = new Rectangle(0, 0, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
    BOUNDS[1] = new Rectangle(DELAY_CHART_WIDTH, 0, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
    BOUNDS[2] = new Rectangle(0, DELAY_CHART_HEIGHT, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
    BOUNDS[3] = new Rectangle(DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
    BOUNDS[4] = new Rectangle(0, 2 * DELAY_CHART_HEIGHT, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
    BOUNDS[5] = new Rectangle(DELAY_CHART_WIDTH, 2 * DELAY_CHART_HEIGHT, DELAY_CHART_WIDTH, DELAY_CHART_HEIGHT);
  }

  static void exportChartAsSVG(List<JFreeChart> charts, File svgFile) {
    DOMImplementation domImpl = GenericDOMImplementation.getDOMImplementation();
    Document document = domImpl.createDocument(null, "svg", null);
    SVGGraphics2D svgGenerator = new SVGGraphics2D(document);
    for (int i = 0; i < charts.size(); ++i) {
      charts.get(i).draw(svgGenerator, BOUNDS[i]);
    }
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream(svgFile);
      Writer out = new OutputStreamWriter(fos, "UTF-8");
      svgGenerator.stream(out, true /* css */);
      out.flush();
    } catch (FileNotFoundException e) {
      throw new RuntimeException(e);
    } catch (UnsupportedEncodingException e) {
      throw new RuntimeException(e);
    } catch (SVGGraphics2DIOException e) {
      throw new RuntimeException(e);
    } catch (IOException e) {
      throw new RuntimeException(e);
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }

  /**
   *
```

```java
 * This method produces the throughput capacity function based↩
    on known hardware configuration, estimated ideal K1
 * and K2 (based on code analysis), and the observations ↩
    derived from experiment simulations (one observation per
 * experiment).
 *
 * @param args
 *            ignored.
 */
public static void main(String args[]) {

  /*
   * Assumptions
   */
  int nMe = 2; // number of microengines running routing ↩
     algorithm in parallel
  int clockMHz = 400; // microengine clock
  int bitsPerPacket = 8 * 66; // each packet is 66 bytes
  int idealK1 = 320; // ideal k1 estimate based on instruction↩
      count of route microcode
  int idealK2 = 88; // ideal k2 estimate based on instruction ↩
     count of route microcode

  /*
   * Observations, one pair (X,Y) per experiment. X is the ↩
      runtime complexity of routing table (iterations of
   * algorithm's inner loop per packet). Y is the throughput ↩
      capacity.
   */
  XYSeries observations = new XYSeries("observed");
  observations.add(0, 692); // experiment 1
  observations.add(2, 508); // experiment 2
  observations.add(7.08, 297); // experiment 3
  observations.add(14.16, 186); // experiment 4
  observations.add(21.24, 168); // experiment 5

  /*
   * compute the throughput capacity chart
   */
  JFreeChart idealOnlyChart = createThroughputCapacityFunction↩
     (false, nMe, clockMHz, bitsPerPacket, idealK1,
      idealK2, observations);
  JFreeChart chart = createThroughputCapacityFunction(true, ↩
     nMe, clockMHz, bitsPerPacket, idealK1, idealK2,
      observations);

    /*
     * export chart to SVG
     */
    List<JFreeChart> charts = new ArrayList<JFreeChart>();
    charts.add(chart);
    exportChartAsSVG(charts, new File("../experiments/throughput_capacity.svg"));
    charts.clear();
    charts.add(idealOnlyChart);
    exportChartAsSVG(charts, new File("../experiments/ideal_throughput_capacity.svg"));
  }
}
```

A.5 IXPSimulationLogAnalyser.java

Listing A.5: IXPSimulationLogAnalyser.java

```java
package thesis;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileFilter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintStream;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.commons.math.stat.descriptive.SummaryStatistics;
import org.apache.commons.math.stat.regression.SimpleRegression;
import org.jfree.chart.JFreeChart;
import org.jfree.data.xy.XYSeries;

import thesis.Packet.TransmittedPacket;
import thesis.RTSetupCodeGenerator.RoutingTable;
import thesis.RTSetupCodeGenerator.StatusVariableSubscriptions;

public class IXPSimulationLogAnalyser {

  Map<String, SummaryStatistics> statistics;
  Map<String, SimpleRegression> regressions;
  List<Packet> packets;
  RoutingTable routingTable;
  double routingTableRuntimeComplexity;

  static final Pattern packetLogLinePattern = Pattern.compile("[ ]*([0-9]+)[ ]([0-9]+)[ ]([0-9]+)"
      + "[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})"
      + "[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})"
      + "[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})"
      + "[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})[ ]([0-9a-f]{8})" + "[ ]([0-9a-f]{4})");

  IXPSimulationLogAnalyser(File experimentFolder, RoutingTable rt) {
    routingTable = rt;
    parsePacketLogs(experimentFolder);

    // compute statistics
    computeForwardingDecisionsAndRoutingTableRuntimeComplexity();
    computeAvgPacketDelay();
    computeLast5AvgDelay();
    computeSummaryStatistics();
    computeLinearRegressions();
  }

  void parsePacketLogs(File experimentFolder) {
    // file handles
    File fileD0P0Rx = new File(experimentFolder, "Device0_Port0_Rx.log");
    File fileD0P1Rx = new File(experimentFolder, "Device0_Port1_Rx.log");
    File fileD0P2Rx = new File(experimentFolder, "Device0_Port2_Rx.log");
    File fileD1P0Tx = new File(experimentFolder, "Device1_Port0_Tx.log");
    File fileD1P1Tx = new File(experimentFolder, "Device1_Port1_Tx.log");
    File fileD1P2Tx = new File(experimentFolder, "Device1_Port2_Tx.log");

    // temporary packet lists.
    List<Packet> port0Rx = new ArrayList<Packet>();
    List<Packet> port1Rx = new ArrayList<Packet>();
    List<Packet> port2Rx = new ArrayList<Packet>();

    // parse files
    if (fileD0P0Rx.isFile()) {
      parseReceiveFile(fileD0P0Rx, 0, port0Rx);
    }
    if (fileD0P1Rx.isFile()) {
      parseReceiveFile(fileD0P1Rx, 1, port1Rx);
    }
    if (fileD0P2Rx.isFile()) {
      parseReceiveFile(fileD0P2Rx, 2, port2Rx);
    }
    if (fileD1P0Tx.isFile()) {
      parseTransmitFile(fileD1P0Tx, 0, port2Rx, port1Rx);
    }
    if (fileD1P1Tx.isFile()) {
      parseTransmitFile(fileD1P1Tx, 1, port0Rx, port2Rx);
    }
    if (fileD1P2Tx.isFile()) {
      parseTransmitFile(fileD1P2Tx, 2, port1Rx, port0Rx);
    }

    createPacketList(port0Rx, port1Rx, port2Rx);
  }

  void createPacketList(List<Packet> port0Rx, List<Packet> port1Rx, List<Packet> port2Rx) {
    // reorganize packets into a single list sorted by packet arrival and
    // port
    packets = new ArrayList<Packet>();
    for (Packet p : port0Rx) {
      packets.add(p);
    }
    for (Packet p : port1Rx) {
      packets.add(p);
    }
    for (Packet p : port2Rx) {
      packets.add(p);
    }
    // sort by least significant sorting criteria first
    Collections.sort(packets, new Comparator<Packet>() {
      @Override
      public int compare(Packet arg0, Packet arg1) {
        return (int) (arg0.rx_port - arg1.rx_port);
      }
    });
    Collections.sort(packets, new Comparator<Packet>() {
      @Override
      public int compare(Packet arg0, Packet arg1) {
        return (int) (arg0.rx_eop - arg1.rx_eop);
      }
    });
    // sort by most significant sorting criteria last
    Collections.sort(packets, new Comparator<Packet>() {
      @Override
      public int compare(Packet arg0, Packet arg1) {
        return (int) (arg0.rx_sop - arg1.rx_sop);
      }
    });

    // assign packet numbers (correlating with arrival time)
    int packet = 1;
    for (Packet p : packets) {
      p.packet = packet++;
    }

    // trim list of packets to contain at most 1000 packets
    while (packets.size() > 1000) {
      packets.remove(packets.size() - 1);
    }
  }

  void parseReceiveFile(File f, int port, List<Packet> list) {
```

```java
    FileInputStream fis = null;
    try {
      fis = new FileInputStream(f);
      BufferedReader br = new BufferedReader(new InputStreamReader(fis, Charset.forName("US-ASCII")));
      while (br.ready()) {
        String line = br.readLine();
        Matcher m = packetLogLinePattern.matcher(line);
        if (m.matches()) {
          if (port == 2 && m.group(1).equals("212")) {
            toString();
          }
          Packet p = new Packet();
          p.packetData = m.group(4) + m.group(5) + m.group(6) + m.group(7) + m.group(8) + m.group(9)
               + m.group(10) + m.group(11) + m.group(12) + m.group(13) + m.group(14) + m.group(15)
               + m.group(16) + m.group(17) + m.group(18) + m.group(19).substring(0, 4);
          p.rx_port = port;
          p.rx_packetNum = Integer.parseInt(m.group(1));
          p.rx_sop = Integer.parseInt(m.group(2));
          p.rx_eop = Integer.parseInt(m.group(3));
          p.timestamp = Long.parseLong(m.group(12).substring(4) + m.group(13) + m.group(14).substring(0, 4),
              16);
          p.variableId = Integer.parseInt(m.group(14).substring(4) + m.group(15).substring(0, 4), 16);
          list.add(p);
        }
      }
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } catch (IOException e) {
      e.printStackTrace();
    } finally {
      if (fis != null) {
        try {
          fis.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }
```

```java

// Uninitialized memory byte patterns
static final String UIP_4 = "5a5a5a5a"; // 4-byte pattern
static final String UIP_2 = "5a5a"; // 2-byte pattern
static final String UIP_PACKET = UIP_4 + UIP_4 + UIP_4 + UIP_4
    + UIP_4 + UIP_4 + UIP_4 + UIP_4 + UIP_4 + UIP_4
    + UIP_4 + UIP_4 + UIP_4 + UIP_4 + UIP_4 + UIP_2;

void parseTransmitFile(File f, int port, List<Packet> source1,
    List<Packet> source2) {
  FileInputStream fis = null;
  try {
    Iterator<Packet> s1It = source1.iterator();
    Iterator<Packet> s2It = source2.iterator();
    fis = new FileInputStream(f);
    BufferedReader br = new BufferedReader(new
        InputStreamReader(fis, Charset.forName("US-ASCII")));
    int s1 = 0;
    int s2 = 0;
    while (br.ready()) {
      String line = br.readLine();
      Matcher m = packetLogLinePattern.matcher(line);
      if (m.matches()) {
        if (port == 1 && m.group(1).equals("424")) {
          toString();
        }
        Packet p;
        String packetData = m.group(4) + m.group(5) + m.group
            (6) + m.group(7) + m.group(8) + m.group(9)
             + m.group(10) + m.group(11) + m.group(12) + m.
                group(13) + m.group(14) + m.group(15)
             + m.group(16) + m.group(17) + m.group(18) + m.
                group(19).substring(0, 4);
        if (UIP_PACKET.equals(packetData)) {
          /*
           * The current implementation of multicasting
              translates copied packet handles to
              uninitialized
           * packet buffers in DRAM. These areas can be
              recognized by repeating bytes of value 0x5a.
           * Because we only use 3 ports we don't need the
              actual packet data to know where it came from,
           * because any incoming packet can only be forwarded
               to at most two other ports, and one of
           * those ports will translate the packet handle to the correct DRAM area and get actual packet
           * data rather than 0x5a patterns.
           */
          p = s2It.next();
          s1++;
          int rxPort = (port + 1) % 3;
          if (rxPort != p.rx_port) {
            throw new RuntimeException("rxPort != p.rx_port, rxPort=" + rxPort + ", p.rx_port="
               + p.rx_port + ", tx_packetNum=" + m.group(1));
          }
          if (s1 != p.rx_packetNum) {
            throw new RuntimeException("s1 != p.rx_packetNum, s1=" + s1 + ", p.rx_packetNum="
               + p.rx_packetNum);
          }
        } else {
          p = s1It.next();
          s2++;
          int rxPort = (port + 2) % 3;
          if (rxPort != p.rx_port) {
            throw new RuntimeException("rxPort != p.rx_port, rxPort=" + rxPort + ", p.rx_port="
               + p.rx_port + ", tx_packetNum=" + m.group(1));
          }
          if (s2 != p.rx_packetNum) {
            throw new RuntimeException("s2 != p.rx_packetNum, s2=" + s2 + ", p.rx_packetNum="
               + p.rx_packetNum);
          }
          if (!packetData.equals(p.packetData)) {
            throw new RuntimeException("tx_port=" + port + ", rx_port=" + p.rx_port + ", tx_packetNum="
               + m.group(1) + ", rx_packetNum=" + p.rx_packetNum
               + ", !packetData.equals(p.packetData), packetData=\"" + packetData
               + "\", p.packetData=\"" + p.packetData + "\"");
          }
        }
        if (p.tp[port] != null) {
          throw new RuntimeException("Packet " + p.rx_packetNum + " from port " + p.rx_port
                + " already has a TransmittedPacket for port " +↩
                  port
                + ". Current transmit packetNum is " + Integer.↩
                 parseInt(m.group(1)));
        }
        p.tp[port] = new TransmittedPacket();
        p.tp[port].tx_sop = Integer.parseInt(m.group(2));
        p.tp[port].tx_eop = Integer.parseInt(m.group(3));
      } else {
        toString();
      }
    }
  } catch (FileNotFoundException e) {
    e.printStackTrace();
  } catch (IOException e) {
    e.printStackTrace();
  } finally {
    if (fis != null) {
      try {
        fis.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}

void ↩
   computeForwardingDecisionsAndRoutingTableRuntimeComplexity↩
   () {
  int pubint = 10;
  int innerLoopIterations = 0;
  for (Packet p : packets) {
    StatusVariableSubscriptions svs = routingTable.find(p.↩
       variableId);
    /*
     * this is basically an implementation of the GridStat ↩
        routing filtering algorithm with additional
     * statistics to estimate the routing table runtime ↩
        complexity for this simulation run
     */
    for (int port = 0; port < 3; ++port) {
      if (p.tp[port] != null) {
        for (int i = 0; i < svs.n[port]; ++i) {
          innerLoopIterations++;
          if ((p.timestamp + (pubint / 2)) % svs.divisors[port↩
            ][i] < pubint) {
            p.tp[port].forwarded = true; // packet should be
            // forwarded to port
            break;
          }
        }
      }
    }
  }
  routingTableRuntimeComplexity = innerLoopIterations / (↩
     double) packets.size();
}

void computeAvgPacketDelay() {
  for (Packet p : packets) {
    int nTransmitted = 0;
    double sumTxDelay = 0;
    for (int i = 0; i < 3; ++i) {
      if (p.tp[i] != null && p.tp[i].forwarded) {
        sumTxDelay += p.tp[i].tx_sop - p.rx_eop;
        ++nTransmitted;
      }
    }
    if (nTransmitted > 0) {
      p.avgTxDelay = sumTxDelay / nTransmitted;
    }
  }
}

void computeLast5AvgDelay() {
  LinkedList<Packet> last5Packets = new LinkedList<Packet>();
  for (Packet p : packets) {
    if (p.avgTxDelay != null) {
      last5Packets.add(p);
      if (last5Packets.size() > 5) {
        last5Packets.removeFirst();
      }
    }
    int nTransmitted = 0;
    double sumTxDelay = 0;
    if (p.avgTxDelay != null && last5Packets.size() >= 5) {
      for (Packet a : last5Packets) {
        sumTxDelay += a.avgTxDelay;
        ++nTransmitted;
      }
      p.avgLast5AvgTxDelay = sumTxDelay / nTransmitted;
    }
  }
}

void computeSummaryStatistics() {
  statistics = new LinkedHashMap<String, SummaryStatistics>();
  computeAvgDelaySS();
  computeAvgLast5DelaySS();
  computeIncomingInterPacketDelaySS();
  computeAddedInterPacketDelayJitterSS();
}

void computeAvgDelaySS() {
  SummaryStatistics avgDelaySS = new SummaryStatistics();
  for (Packet p : packets) {
    if (p.avgTxDelay != null) {
      avgDelaySS.addValue(p.avgTxDelay);
    }
  }
  statistics.put("AverageDelay", avgDelaySS);
}

void computeAvgLast5DelaySS() {
  SummaryStatistics avglast5DelaySS = new SummaryStatistics();
  for (Packet p : packets) {
    if (p.avgLast5AvgTxDelay != null) {
      avglast5DelaySS.addValue(p.avgLast5AvgTxDelay);
    }
  }
  statistics.put("AverageLast5AverageDelay", avglast5DelaySS);
}

void computeIncomingInterPacketDelaySS() {
  SummaryStatistics rxInterPacketDelaySS = new ↩
     SummaryStatistics();
  Packet prevIncomingPacket[] = new Packet[3];
  for (Packet p : packets) {
    if (prevIncomingPacket[p.rx_port] != null) {
     /*
         * only look at the packets after the first 100 because ↩
          the first few packets possibly arrive faster
         * than the configured throughput dictate to compensate ↩
          for the time it takes to initialize the hardware
```

```java
         * and memory layout.
         */
        if (p.packet > 100) {
          rxInterPacketDelaySS.addValue(p.rx_eop - ↩
             prevIncomingPacket[p.rx_port].rx_eop);
        }
      }
      prevIncomingPacket[p.rx_port] = p;
    }
    statistics.put("IncomingInterPacketDelay", ↩
       rxInterPacketDelaySS);
  }

  void computeAddedInterPacketDelayJitterSS() {
    SummaryStatistics txInterPacketDelaySS = new ↩
       SummaryStatistics();
    Packet prevIncomingPacket[] = new Packet[3];
    for (Packet p : packets) {
      if (prevIncomingPacket[p.rx_port] != null) {
        /*
         * only look at the packets after the first 100 because ↩
            the first few packets possibly arrive faster
         * than the configured throughput dictate to compensate ↩
            for the time it takes to initialize the hardware
         * and memory layout.
         */
        if (p.packet > 100) {
          int d1 = (p.rx_port + 1) % 3; // first destination (↩
             port)
          int d2 = (p.rx_port + 2) % 3; // second destination (↩
             port)
          if (p.tp[d1] != null && prevIncomingPacket[p.rx_port].↩
             tp[d1] != null) {
            txInterPacketDelaySS.addValue(p.tp[d1].tx_eop - ↩
               prevIncomingPacket[p.rx_port].tp[d1].tx_eop);
          }
          if (p.tp[d2] != null && prevIncomingPacket[p.rx_port].↩
             tp[d2] != null) {
            txInterPacketDelaySS.addValue(p.tp[d2].tx_eop - ↩
               prevIncomingPacket[p.rx_port].tp[d2].tx_eop);
          }
        }
      }
      prevIncomingPacket[p.rx_port] = p;
    }
    statistics.put("OutgoingInterPacketDelay", ↩
       txInterPacketDelaySS);
  }

  void computeLinearRegressions() {
    regressions = new LinkedHashMap<String, SimpleRegression>();
    computeLinearRegressionOfPacketDelayAsFunctionOfTime();
    computeLinearRegressionOfLastHalfPacketDelayAsFunctionOfTime↩
       ();
  }

  void computeLinearRegressionOfPacketDelayAsFunctionOfTime() {
    SimpleRegression sr = new SimpleRegression();
    for (Packet p : packets) {
      for (int port = 0; port < 3; ++port) {
        if (p.tp[port] != null) {
          double y = p.tp[port].tx_sop - p.rx_eop;
          double x = p.packet;
          sr.addData(x, y);
        }
      }
    }
    regressions.put("PacketDelayAsFunctionOfTime", sr);
  }

  void ↩
     computeLinearRegressionOfLastHalfPacketDelayAsFunctionOfTime↩
     () {
    SimpleRegression sr = new SimpleRegression();
    for (Packet p : packets) {
      if (p.packet >= packets.size() / 4) {
        for (int port = 0; port < 3; ++port) {
          if (p.tp[port] != null) {
            double y = p.tp[port].tx_sop - p.rx_eop;
            double x = p.packet;
            sr.addData(x, y);
          }
        }
      }
    }
    regressions.put("TrimmedPacketDelayAsFunctionOfTime", sr);
  }

  void printCsv(File outputFile) {
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream(outputFile);
      PrintStream out = new PrintStream(fos);
      out.println(Packet.csvHeader);
      for (Packet p : packets) {
        out.println(p.toCsv());
      }
      out.flush();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
  }

  static final Pattern experimentFolderPattern = Pattern.compile↩
     ("experiment([1-9][0-9]*)");
  static final Pattern simulationRunFolderPattern = Pattern.↩
     compile("3x([1-9][0-9]*)");

  static class SingleSimulationRun {
    int simulationId;
    int incomingThroughput;
    File f;
    IXPSimulationLogAnalyser logAnalyzer;
  }

  static void printCrossSummaryTexJitterTable(File f, List<↩
     SingleSimulationRun> simulations) {
    FileOutputStream fos = null;
    try {
      fos = new FileOutputStream(f);
      PrintStream out = new PrintStream(fos);
      out.println("\\begin{table}");
      out.println("\\caption{Incoming and outgoing inter-packet-↩
         delay per simulation of experiment \\theex}");
      out.println("\\centering");
      out.println("\\begin{tabular}{|c|c|c c|c c|}");
      out.println("\\hline");
      out.println("$s$ & Input & Input Average & StdDev & Output Average & StdDev \\\\");
      out.println(" & (Mbps) & Inter-packet delay & & Inter-packet delay & \\\\");
      out.println(" & & (media-bus cycles) & & (media-bus cycles) & \\\\");
      out.println("\\hline");
      for (SingleSimulationRun ssr : simulations) {
        Matcher m = simulationRunFolderPattern.matcher(ssr.f.getName());
        m.matches();
        int fractionOfThroughput = Integer.parseInt(m.group(1));
        SummaryStatistics incomingInterPacketDelaySS = ssr.logAnalyzer.statistics
            .get("IncomingInterPacketDelay");
        SummaryStatistics outgoingInterPacketDelaySS = ssr.logAnalyzer.statistics
            .get("OutgoingInterPacketDelay");
        out.format("\\thesimc \\addtocounter{simc}{1} & %d & %g & %g & %g & %g\\\\", 3 * fractionOfThroughput,
            incomingInterPacketDelaySS.getMean(), incomingInterPacketDelaySS.getStandardDeviation(),
            outgoingInterPacketDelaySS.getMean(), outgoingInterPacketDelaySS.getStandardDeviation());
        out.println();
      }
      out.println("\\hline");
      out.println("\\end{tabular}");
      out.println("\\label{table:se\\theex}");
      out.println("\\end{table}");
      out.println("\\endinput");
      out.flush();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
    } finally {
      if (fos != null) {
        try {
          fos.close();
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
```

```
  }

  /**
   * Usage: java thesis.IXPSimulationLogAnalyser <experiment ↩
      folder> [<status variable id> <port> <subscriber
   * rate >]...
   *
   */
  public static void main(String[] args) {

    // parse program arguments
    File experimentFolder = new File(args[0]);
    int nextSimulationId = Integer.parseInt(args[1]);
    List<Integer> list = new ArrayList<Integer>();
    for (int i = 2; i < args.length; i += 3) {
      list.add(new Integer(args[i]));
      list.add(new Integer(args[i + 1]));
      list.add(new Integer(args[i + 2]));
    }

    // setup internal routing table which is the same across ↩
       simulations
    // within an experiment
    RoutingTable rt = RTSetupCodeGenerator.createRoutingTable(↩
       list);

    // find experiment folder sub directories which corresponds ↩
       with
    // simulation runs
    File[] subdirs = experimentFolder.listFiles(new FileFilter()↩
        {
      @Override
      public boolean accept(File f) {
        Matcher m = simulationRunFolderPattern.matcher(f.getName↩
           ());
        return m.matches() && f.isDirectory();
      }
    });

    // parse and analyze input files
    boolean problem = false;
    List<SingleSimulationRun> simulations = new LinkedList<↩
       SingleSimulationRun>();
    for (File f : subdirs) {
      try {
```

```java
        SingleSimulationRun ssr = new SingleSimulationRun();
        ssr.f = f;
        ssr.logAnalyzer = new IXPSimulationLogAnalyser(f, rt);
        simulations.add(ssr);
      } catch (Exception e) {
        problem = true;
        try {
          System.err.println("Exception while analysing folder " + f.getCanonicalPath());
        } catch (IOException e1) {
          e1.printStackTrace();
        }
        e.printStackTrace();
        break;
      }
    }
    // sort simulation list ascending by throughput
    Collections.sort(simulations, new Comparator<SingleSimulationRun>() {
      @Override
      public int compare(SingleSimulationRun o1, SingleSimulationRun o2) {
        Matcher m1 = simulationRunFolderPattern.matcher(o1.f.getName());
        m1.matches();
        int t1 = Integer.parseInt(m1.group(1));
        Matcher m2 = simulationRunFolderPattern.matcher(o2.f.getName());
        m2.matches();
        int t2 = Integer.parseInt(m2.group(1));
        return t1 - t2;
      }
    });

    // assign simulation id and throughput
    for (SingleSimulationRun ssr : simulations) {
      ssr.simulationId = nextSimulationId++;
      Matcher m = simulationRunFolderPattern.matcher(ssr.f.getName());
      m.matches();
      ssr.incomingThroughput = 3 * Integer.parseInt(m.group(1));
    }

    // write output files
    if (!problem) {
      Matcher m = experimentFolderPattern.matcher(↩
         experimentFolder.getName());
      int experimentNo = 0;
      if (m.matches()) {
        experimentNo = Integer.parseInt(m.group(1));
      }
      File outFolder = experimentFolder.getParentFile();
      String ePrefix = experimentFolder.getName() + "_";

      // print new packet logs for all simulations in experiment
      for (SingleSimulationRun ssr : simulations) {
        @SuppressWarnings("unused")
        String prefix = ePrefix + ssr.f.getName() + "_";
        // ssr.logAnalyzer.printCsv(new File(outFolder, prefix +↩
            "packets.csv"));
      }

      // Generate delay charts
      List<JFreeChart> charts = new ArrayList<JFreeChart>();
      for (SingleSimulationRun ssr : simulations) {
        JFreeChart chart = GraphProducer.↩
           createDelayWithLinearFit(ssr.logAnalyzer.packets,
            ssr.logAnalyzer.regressions.get("↩
               PacketDelayAsFunctionOfTime"), "Simulation "
               + ssr.simulationId + ", " + ssr.↩
                  incomingThroughput + " Mbps",
            (experimentNo == 5) ? 5000 : 0);
        charts.add(chart);
      }
      // render SVG graphs in groups of up to 6 per page
      for (int i = 0; i < charts.size(); i += 6) {
        GraphProducer.exportChartAsSVG(charts.subList(i, Math.↩
           min(i + 6, charts.size())), new File(outFolder,
            ePrefix + "delay_" + (i / 6) + ".svg"));
      }

      // Generate slope chart
      XYSeries slopeSeries = new XYSeries("slope");
      for (SingleSimulationRun ssr : simulations) {
        slopeSeries.add(ssr.incomingThroughput, ssr.logAnalyzer.↩
           regressions.get("PacketDelayAsFunctionOfTime")
            .getSlope());
      }
      JFreeChart slopeChart = GraphProducer.createSlopeOfDelay(↩
         slopeSeries);
      charts.clear();
      charts.add(slopeChart);
      GraphProducer.exportChartAsSVG(charts, new File(outFolder,↩
          ePrefix + "slope.svg"));

      // print experiment cross simulation summary
      printCrossSummaryTexJitterTable(new File(outFolder, ↩
         experimentFolder.getName() + "_jitter.tex"),
          simulations);
    }
  }
}
```

A.6 utilities.uc

Listing A.6: utilities.uc

```


//
//  C = A * B
//
// 16 instructions
#macro imul64x32[c2, c1, c0, a1, a0, b0]
.begin
  .reg hi32

  // C_high64 = A_high32 * B
  mul_step[a1, b0], 32x32_start
  mul_step[a1, b0], 32x32_step1
  mul_step[a1, b0], 32x32_step2
  mul_step[a1, b0], 32x32_step3
  mul_step[a1, b0], 32x32_step4
  mul_step[c1, --], 32x32_last
  mul_step[c2, --], 32x32_last2

  // C_low64 = A_low32 * B
  mul_step[a0, b0], 32x32_start
  mul_step[a0, b0], 32x32_step1
  mul_step[a0, b0], 32x32_step2
  mul_step[a0, b0], 32x32_step3
  mul_step[a0, b0], 32x32_step4
  mul_step[c0, --], 32x32_last
  mul_step[hi32, --], 32x32_last2

  // C_mid32 = (C_low64 >> 32) + (C_high64 & 0xffffffff)
  alu[c1, c1, +, hi32]

  // C_high32 = (C_high64 >> 32) + carry_out from computing ↩
     C_mid32
  alu[c2, c2, +carry, 0]

.end
#endm

//
//
//  A is a 64 bit integer described by two 32 bit registers,
//  a1 is the high 32 bits of A, and a0 is the low 32 bits of A.
//
//  Q = (A * Reciprocal) >> (32+nbits)
//  rem = A - Q * divisor
//  while ( Rem >= divisor )
//    Rem -= divisor
//
//
#macro remainder64x32[rem, a1, a0, divisor, reciprocal, nbits]
.begin
  .reg q1 q0 c0 kd
  .reg c2 c1 remh

  //  Q = (A * Reciprocal) >> (32+nbits)

  // Q = A * Reciprocal
  imul64x32[q1, q0, c0, a1, a0, reciprocal]

  // q0 = q0 >> nbits
  alu[--, nbits, OR, 0]
  alu[q0, --, B, q0, >>indirect]

  // q0 = q0 | (q1 << (32-nbits))
  alu[c0, 32, -, nbits]
  alu[--, c0, OR, 0]
  alu[q0, q0, OR, q1, <<indirect]

  // q1 = q1 >> nbits
  alu[--, nbits, OR, 0]
  alu[q1, --, B, q1, >>indirect]
```

```

// 23 instructions


// Q is now a 64 bit integer, with the high
// 32 bits at q1 and low 32 bits at q0

// rem = A - (Q * divisor)

// C = Q * divisor

imul64x32[c2, c1, c0, q1, q0, divisor]

// FIXME: fix for the case where the high 32 bits in A and Q *↩
    divisor
//        are not the same.
// we here assume that the high 32 bits in both A and (Q * ↩
   divisor)
// are the same, hence we only need to look at the low 32 bits↩
   .
//mul_step[q0, divisor], 32x32_start
//mul_step[q0, divisor], 32x32_step1
//mul_step[q0, divisor], 32x32_step2
//mul_step[q0, divisor], 32x32_step3
//mul_step[q0, divisor], 32x32_step4
//mul_step[c0, --], 32x32_last
//mul_step[c1, --], 32x32_last2 ; discard

alu[rem, a0, -, c0]
alu[remh, a1, -carry, c1] ; remh should be 0

  .if ( c1 != a1 )
      nop; should never happen but included here to enable ↩
         debug breakpoint
  .endif

// 30 instructions

// at this point the remainder might be up to 3*divisor off ↩
   from the
// actual remainder, so iterate until remainder is less than ↩
   divisor.

// NOTE: for some reason the remainder ends up being k*divisor↩
    off from
  // the actual remainder, so just subtract k*divisor from rem.

  alu_shf[kd, --, B, divisor, <<5]
  .while ( rem >= kd )
    alu[rem, rem, -, kd]
  .endw

  alu_shf[kd, --, B, divisor, <<4]
  .if ( rem >= kd )
    alu[rem, rem, -, kd]
  .endif

  alu_shf[kd, --, B, divisor, <<3]
  .if ( rem >= kd )
    alu[rem, rem, -, kd]
  .endif

  alu_shf[kd, --, B, divisor, <<2]
  .if ( rem >= kd )
    alu[rem, rem, -, kd]
  .endif

  alu_shf[kd, --, B, divisor, <<1]
  .if ( rem >= kd )
    alu[rem, rem, -, kd]
  .endif

  .if ( rem >= divisor )
    alu[rem, rem, -, divisor]
  .endif

.end
#endm
```

A.7 routing_table.uc

Listing A.7: routing_table.uc

```
#include <utilities.uc>

.sig volatile sig_rtable

#define CAM_VALID 1
```

```
#define CAM_INVALID 0

#macro hash_to_lwoffset[ lwoffset, in_hash ]
  .begin
#if ( GRIDSTAT_HTABLE_SIZE == 128 )
  // lwoffset = in_hash & 0x7F
  alu[lwoffset, in_hash, AND, 0x7F]
#elif ( GRIDSTAT_HTABLE_SIZE == 256 )
  // lwoffset = in_hash & 0xFF
  ld_field_w_clr[lwoffset, 0001, in_hash]
#elif ( GRIDSTAT_HTABLE_SIZE == 512 )
  // lwoffset = in_hash & 0x1FF
  ld_field_w_clr[lwoffset, 0011, in_hash]
  alu_shf[lwoffset, lwoffset, ~AND, 0xFE, <<8]
#elif ( GRIDSTAT_HTABLE_SIZE == 1024 )
  // lwoffset = in_hash & 0x3FF
  ld_field_w_clr[lwoffset, 0011, in_hash]
  alu_shf[lwoffset, lwoffset, ~AND, 0xFC, <<8]
#else
#error "GRIDSTAT_HTABLE_SIZE must be one of (128, 256, 512, ↩
   1024)"
#endif
  .end
#endm


#macro getrtaddr[ sram_offset, variableid ]
.begin
  .reg varid1 varid2 varid3
  move( varid1, 1 )
  move( varid2, 2 )
  move( varid3, 3 )
  // get base offset
  .if ( variableid == varid1 )
    move( sram_offset, (GRIDSTAT_RTABLE_BASE+GRIDSTAT_RTENT_SIZE↩
       ) )
  .elif ( variableid == varid2 )
    move( sram_offset, (GRIDSTAT_RTABLE_BASE+2*↩
       GRIDSTAT_RTENT_SIZE) )
  .elif ( variableid == varid3 )
    move( sram_offset, (GRIDSTAT_RTABLE_BASE+3*↩
       GRIDSTAT_RTENT_SIZE) )
  .else
    // default entry if no variableid match
    move( sram_offset, (GRIDSTAT_RTABLE_BASE+0) ) // use varid 0 which corresponds to no entry
  .endif
.end
#endm

// output sram_offset will be 0 if no entry matching variableid can be found
#macro init_rtable_offset[ variableid ]
.begin
  .reg sram_offset lsmask tmp cam_res lru lm_a i

  // lockup variableid in cam
  cam_lookup[cam_res, variableid]

  // check for hit or miss, if hit, then we don't need to load rt data from sram
  alu_shf[--, 0x1, AND, cam_res, >>7]
  bne[rt_entry_in_lm#]

  // cam lookup miss, load routing table entry for variableid into local memory
  alu_shf[lru, 0xf, AND, cam_res, >>3] ; find least recently used entry
  cam_write[lru, variableid, CAM_INVALID] // update cam

    /* The following commented section uses the hashing instruction to lookup
     * routing table entry in a hashtable fashion, it is ommitted for simplicity.
     * The following commented code section hasn't been properly tested and may
     * contain bugs.
     */

    /*
    xbuf_alloc($hash_data, 2, read_write)

  move[$hash_data0, variableid]
  move[$hash_data1, 0]

  hash_48[$hash_data0, 1], sig_done[sig_rtable]

  // wait for hash io, there are 2 signals to wait for
  alu[lsmask, --, B, 0]
  alu[tmp, --, B, &sig_rtable]
  alu[--, tmp, OR, 0]
  alu_shf[lsmask, lsmask, OR, 1, <<indirect]
  alu[tmp, tmp, +, 1]
  alu[--, tmp, OR, 0]
  alu_shf[lsmask, lsmask, OR, 1, <<indirect]
  local_csr_wr[active_ctx_wakeup_events, lsmask]
  ctx_arb[--]
  .io_completed sig_rtable

    hash_to_lwoffset[sram_offset, $hash_data0]
  // TODO: include multiple mode functionality
  add_shf_left(sram_offset, GRIDSTAT_HTABLE_BASE, sram_offset, ↩
     2) ; <<2 to convert sram_offset to byte offset
  sram[read, $hash_data0, sram_offset, 0, 1], ctx_swap[↩
     sig_rtable] ; get address of first entry in list

  // The following loop iterates through the single linked list
  // until variableid matches one of the entries
htable_list_iterate#:
  alu[sram_offset, --, B, $hash_data0]
  beq[htable_list_iterate_end#] // if there is no rt entry, stop↩
      iterating
  sram[read, $hash_data0, sram_offset, 0, 2], ctx_swap[↩
     sig_rtable]
  alu[--, variableid, -, $hash_data1]
  bne[htable_list_iterate#]
htable_list_iterate_end#:

  xbuf_free($hash_data)
    */

  // for now the routing table is hardcoded, so find the actual ↩
     sram address
  getrtaddr[sram_offset, variableid]

  // advance pointer to start of subintervals
  alu[sram_offset, sram_offset, +, 16]

  xbuf_alloc($filter_data, 6, read)

  // size of one routing table entry is 36 long words
  ; lm_a = lru * 36
  alu_shf[lm_a, --, B, lru, <<5]   ; lm_a = lru * 32
  add_shf_left(lm_a, lm_a, lru, 2) ; lm_a = lm_a + lru * 4
  alu_shf[lm_a, --, B, lm_a, <<2]  ; move to offset used ↩
      ACTIVE_LM_ADDR_0 write

  local_csr_wr[ACTIVE_LM_ADDR_0, lm_a]

  alu[i, --, B, 0]
  .while ( i < 6 )

    sram[read, $filter_data0, sram_offset, 0, 6], sig_done[↩
        sig_rtable]

    alu[i, i, +, 1]
    alu[sram_offset, sram_offset, +, 24]

    // wait for previous sram read from routing table
    ctx_arb[sig_rtable]

    alu[*l$index0++, --, B, $filter_data0]
    alu[*l$index0++, --, B, $filter_data1]
    alu[*l$index0++, --, B, $filter_data2]
    alu[*l$index0++, --, B, $filter_data3]
    alu[*l$index0++, --, B, $filter_data4]
    alu[*l$index0++, --, B, $filter_data5]

  .endw

  cam_write_state[lru, CAM_VALID] // set cam entry as valid

  xbuf_free($filter_data)

rt_entry_in_lm#:

.end
#endm

#macro get_stream_info[pubint]
.begin
  // TODO: download value of pubint from routing table
  alu[pubint, --, B, 10]
.end
#endm

//
//  Assumes that there is an array of FILTER_ENTRY starting
//  at sram_addr. The array is terminated when (element[i].divisor == 0)
//
//
//  typedef struct s_filter_entry {
//    unsigned long divisor;
//    unsigned long reciprocal;
//  } FILTER_ENTRY;
//
//  bits 27:31 of divisor contains the number of significant bits in
//  reciprocal, while bits 0:26 contains the divisor itself.
//
//
//
#macro dl_gridstat_filter[ forward, out_port, variableid, timestamp1, timestamp0, pubint]
.begin
  .reg divisor reciprocal nbits rem sram_offset txreg_addr cam_res cam_e lm_a i

  br[try_cam_valid#]

wait_cam_load#:
  ctx_arb[voluntary]

try_cam_valid#:
  cam_lookup[cam_res, variableid]
  alu_shf[tmp, 0x1, AND, cam_res, >>7]
  sub_shf_right(tmp, CAM_VALID, cam_res, 8) // extract state bits
  bne[wait_cam_load#] ; branch if CAM_VALID is not set

  alu[forward, --, B, 0] // default to dropping packet

  alu_shf[cam_e, 0xf, AND, cam_res, >>3] ; find cam entry
  ; lm_a = cam_e * 36
  alu_shf[lm_a, --, B, cam_e, <<5]    ; lm_a = cam_e * 32
  add_shf_left(lm_a, lm_a, cam_e, 2) ; lm_a = lma_a + cam_e * 4

  .if ( out_port == 1 )
    alu[lm_a, lm_a, +, 12]
  .elif ( out_port == 2 )
    alu[lm_a, lm_a, +, 24]
  .endif

  alu_shf[lm_a, --, B, lm_a, <<2]  ; move to offset used ACTIVE_LM_ADDR_0 write

    //alu[i, --, B, 1]
    //.while (i>0)
  alu[forward, --, B, 0] // default to dropping packet
  local_csr_wr[ACTIVE_LM_ADDR_0, lm_a]
  nop
  nop
  nop

  // The following loop will terminate if a decision to forward the packed
  // is reached, or when we get to the end of the filter_entry array.
  .while( *l$index0 && !forward )

    alu[tmp, --, B, *l$index0++]
    alu_shf[nbits, --, B, tmp, >>27]
    alu_shf[divisor, tmp, AND~, 0x1F, <<27]
    alu[reciprocal, --, B, *l$index0++]

    // if ( (timestamp + pubint/2) % divisor < pubint ) then FORWARD (i.e. don't drop)
    // Note that the timestamp should already have pubint/2 added to it.

    remainder64x32[rem, timestamp1, timestamp0, divisor, reciprocal, nbits]

    .if ( rem < pubint )
      // FORWARD PACKET
      alu[forward, --, B, 1]
    .endif

  .endw

    // alu[i, i, -, 1]
    //.endw

    alu[forward, --, B, 1] // forward packet

.end
#endm
```

```

#macro rt_next_port[]
.begin
  ;alu[sram_addr, sram_addr, +, (8*8)] ; sram_addr += sizeof(FILTER_ENTRY) * MAX_SUBINTS
.end
#endm
```

A.8 packet_echo.uc

Listing A.8: packet_echo.uc

```
/*
 * This file is the entry point of the Route microblock and implements
 * a simple GridStat status router.
 *
 * This file sources packets from the packet_rx[] microblock and forwards them
 * to the corresponding port transmit queue for the sphy_mphy4_tx[] microblock.
 *
 */

#include <dl_system.h>
#include <stdmac.uc>
#include <hardware.h>
#include <dl_meta.uc>
#include <dl_buf.uc>
#include <dl_misc.uc>
#include <sig_macros.uc>

#include <routing_table.uc>

/*
 * Signal and register definitions
 */

// absolute registers
//.reg  @count_p0
//.reg  @count_p1
//.reg  @count_p2
//.reg  @count_p3
```

```
//.reg   @count_empty
//.reg   @debug_cnt

// External reply interface from packet copier
.reg global volatile read $rx_copy_handle
.sig visible volatile sig_copy_done
.addr sig_copy_done PACKET_COPY_REPLY_SIGNAL

// thread ordering signals and registers
.sig volatile   sig_prev
.addr     sig_prev        DL_SIG_WAKE

// scratch ring put msgs
//xbuf_alloc($tx_pktc, 3, write) ; request msg to packet copier
//xbuf_alloc($txreq, 3, write) ; tx request message to ethernet TX
//xbuf_alloc($$gdata, 4, read)
.reg global volatile write $tx_pktc0 $tx_pktc1 $tx_pktc2
.xfer_order $tx_pktc0 $tx_pktc1 $tx_pktc2
.reg global volatile write $txreq0 $txreq1 $txreq2
.xfer_order $txreq0 $txreq1 $txreq2
.reg global volatile read $$gdata0 $$gdata1 $$gdata2 $$gdata3
.xfer_order $$gdata0 $$gdata1 $$gdata2 $$gdata3

.reg global volatile $rwxf_meta0 $rwxf_meta1 $rwxf_meta2 $rwxf_meta3
.xfer_order $rwxf_meta0 $rwxf_meta1 $rwxf_meta2 $rwxf_meta3
.sig  volatile sig_meta

.reg global volatile sig_mask
.reg global volatile dl_buf_handle    // The current Buffer handle.
.reg global volatile dl_buf_copy_handle   // The current Buffer copy handle.
.reg global volatile dl_eop_buf_handle    // For large packets, this is the last buffer in the chain
.reg global volatile dl_next_block      // Next block that should process the buffer/packet
.reg global volatile port_mask forward t1 t0 varid

// This is a global signal that indicates that
// system initialization is done
.sig  volatile system_init_done
.addr   system_init_done  me_init_signal

```

```
.sig  volatile scr_get scr_put_copy scr_put0 scr_put1 scr_put2 sig_dram; signal for scratch put


#define NUM_PORTS 3

.reg global volatile port[3] // packet handles or UC_NULL

/*
 * The request_packet_copy macro was written to interface
 * with the packet_copier microblock. That microblock and
 * this macro is no longer in use, as we now use a different
 * strategy to enable multicasting.
 */
#macro request_packet_copy[sig_mask]
.begin
  .reg tmp tmp_lw port_out0 port_out1 port_in

  .set dl_meta[1] dl_meta[2] dl_meta[3]

  dl_meta_get_input_port[port_in]

  // Calculate output port (0->1, 1->2, 2->0)
  alu[port_out0, port_in, +, 1]
  .if(port_out0 == 3)
    alu[port_out0, --, b, 0]
  .endif

  // compute output port of target, if source/target input port is 0, the target output port is
  // either 1 or 2 (the one that's not source output port).
  alu[tmp_lw, port_in, +, port_out0]
  alu[port_out1, tmp_lw, XOR, 0x3] // port_out1 = the remaining port

  // Prepare copy request for the packet copier

  //dl_meta_get_packet_size[pktc_packet_size]

  alu[$tx_pktc[0], --, B, dl_buf_handle]

  dl_meta_get_buffer_size[tmp] // use only the first buffer for now
  alu_shf[tmp_lw, --, B, tmp, <<16]
  dl_meta_get_offset[tmp]
  alu[tmp_lw, tmp_lw, OR, tmp]
  alu[$tx_pktc[1], --, B, tmp_lw]

  local_csr_rd[ACTIVE_CTX_STS]
  immed[tmp, 0]
  alu[tmp, tmp, AND, 0x7] // tmp = ctx
  alu_shf[tmp_lw, port_out1, OR, tmp, <<4] // tmp_lw = port_out1 | (ctx << 4)
  move[tmp, __UENGINE_ID] // tmp = ME
  alu_shf[tmp_lw, tmp_lw, OR, tmp, <<7] // tmp_lw = port_out1 | (ctx << 4) | (ME << 7)
  alu_shf[tmp_lw, tmp_lw, OR, &$rx_copy_handle, <<12] // tmp_lw = port_out1 | (ctx << 4) | (ME << 7) | ($rx <<12)
  alu[$tx_pktc[2], --, B, tmp_lw]

  br[pktc_test_ring_full#]

pktc_ring_full#:
  nop
  ctx_arb[voluntary]

pktc_test_ring_full#:
  br_inp_state[SCR_RING/**/PACKET_COPIER_SCR_RING/**/_STATUS, pktc_ring_full#]

  // Send a copy request to the packet copier
  alu_shf[tmp, --, B, PACKET_COPIER_SCR_RING, <<2]
  scratch[put, $tx_pktc[0], 0, tmp, 3], ctx_swap[scr_put_copy]

  // sig_mask |= (1<<sig_copy_done)
  alu[tmp, --, B, &sig_copy_done]
  alu[--, tmp, OR, 0]
  alu_shf[sig_mask, sig_mask, OR, 1, <<indirect]

.end
#endm

#macro pc_get_copy_handle[_in_handle, _out_handle]
.begin
    .reg tmp_buf_sram_size
  move(tmp_buf_sram_size, BUF_SRAM_SIZE);
    alu[_out_handle, _in_handle, +, tmp_buf_sram_size]
.end
#endm

```

```
#macro prepare_port_out_handles[]
.begin
  .reg port_out0 port_out1 port_in tmp

  alu[port[0], --, B, UC_NULL]
  alu[port[1], --, B, UC_NULL]
  alu[port[2], --, B, UC_NULL]

  dl_meta_get_input_port[port_in]

  // Calculate output port (0->1, 1->2, 2->0)
  alu[port_out0, port_in, +, 1]
  .if(port_out0 == 3)
    alu[port_out0, --, b, 0]
  .endif

  // compute output port of target, if source/target input port is 0, the target output port is
  // either 1 or 2 (the one that's not source output port).
  alu[tmp, port_in, +, port_out0]
  alu[port_out1, tmp, XOR, 0x3] // port_out1 = the remaining port


  // port[port_out0] = dl_buf_handle
  alu_shf[tmp, --, b, port_out0, <<1]
  jump[tmp, port_out0_0#], targets[port_out0_0#, port_out0_1#, port_out0_2#]
  port_out0_0#:
    alu[port[0], --, B, dl_buf_handle]
    br[port_out0_end#]
  port_out0_1#:
    alu[port[1], --, B, dl_buf_handle]
    br[port_out0_end#]
  port_out0_2#:
    alu[port[2], --, B, dl_buf_handle]
    //br[port_out0_end#]
  port_out0_end#:


    // inline packet copy

    // Load source packet meta data
  dl_meta_load_cache[dl_buf_handle, $rwxf_meta, sig_meta, 0, 4]

```

```
// Get available packet buffer
  pc_get_copy_handle[dl_buf_handle, dl_buf_copy_handle]

// Wait for source packet meta data to load
ctx_arb[sig_meta]

// Setup local cache in dl_meta GPRs
dl_meta_init_cache[$rwxf_meta[1], $rwxf_meta[1], $rwxf_meta[2], $rwxf_meta[3], 0, 0, 0, 0]

// get input and output port of source packet.
//dl_meta_get_input_port[port_in]
//dl_meta_get_output_port[port_out]

// Change output port of local meta data cache to reflect that of the target packet.
dl_meta_set_output_port[port_out1]

// Write a subset of meta data cache to the target packet, this is correct as
// the target (copy) is to have the same meta data as the source except for
// output port and next buffer/packet pointers.
dl_meta_flush_cache[$rwxf_meta, dl_buf_copy_handle, sig_meta, SIG_NONE, 0, 4]



// port[port_out1] = copied handle // set it to 0 for now which is differenct from UC_NULL (0xFF)
alu_shf[tmp, --, b, port_out1, <<1]
jump[tmp, port_out1_0#], targets[port_out1_0#, port_out1_1#, port_out1_2#]
port_out1_0#:
  alu[port[0], --, B, dl_buf_copy_handle]
  //alu[port[0], --, B, UC_NULL]
  br[port_out1_end#]
port_out1_1#:
  alu[port[1], --, B, dl_buf_copy_handle]
  //alu[port[1], --, B, UC_NULL]
  br[port_out1_end#]
port_out1_2#:
  alu[port[2], --, B, dl_buf_copy_handle]
  //alu[port[2], --, B, UC_NULL]
  //br[port_out1_end#]
  port_out1_end#:

.end
#endm

#macro request_packet_gridstat_data[]
.begin
  .reg sdram_data_base sdram_offset tmp

  dl_buf_get_data[sdram_data_base, dl_buf_handle]

  // set sdram_offset to start of packet data
  dl_meta_get_offset[tmp]
  alu[sdram_offset, --, B, tmp]

  // offset  0, size 14 - Ethernet
  // offset 14, size 20 - IP
  // offset 34, size  8 - UDP (skipped in simulation)
  // offset 42, size xx - Gridstat
  alu[sdram_offset, sdram_offset, +, 34]

  dram[read, $$gdata0, sdram_data_base, sdram_offset, 2], ↩
     sig_done[sig_dram]
.end
#endm

// must call request_packet_gridstat_data before calling this ↩
   function
#macro get_packet_gridstat_data[t1, t0, varid]
.begin

  ctx_arb[sig_dram]

  // NOTE: extracting timestamp and variableid assumes the ↩
     gridstat message
  // starts at offset 42 and uses network byte order (BIG ENDIAN↩
     )

  // GRIDSTAT message format:
  // offset  0, size 8 - timestamp
  // offset  8, size 4 - variable ID

  // offset 42, size 4 - high 32 bits of timestamp
  ld_field_w_clr[t1, 1100, $$gdata0, <<16]
  ld_field[t1, 0011, $$gdata1, >>16]

  // offset 46, size 4 - low 32 bits of timestamp
  ld_field_w_clr[t0, 1100, $$gdata1, <<16]
  ld_field[t0, 0011, $$gdata2, >>16]

  // offset 50, size 4 - 32 bit variable ID
  ld_field_w_clr[varid, 1100, $$gdata2, <<16]
  ld_field[varid, 0011, $$gdata3, >>16]

.end
#endm

#macro dl_gridstat_route[sig_mask]
.begin
  // Make sure current packet is intended for GRIDSTAT routing
  alu[--, dl_next_block, -, BID_GRIDSTAT]
  bne[end_dl_gridstat_route#]

  request_packet_gridstat_data[]

  //request_packet_copy[sig_mask]

  prepare_port_out_handles[]

  // Lookup what ports might want current packet
  alu[port_mask, --, B, 0x7] // all 3 ports

  .reg port_in
  dl_meta_get_input_port[port_in]

  // except incoming port
  bits_clr(port_mask, port_in, 1)


  // Rate filter per port

  .reg cur_port
  alu[cur_port, --, B, 0]

    // 50 instructions

  // wait for dram read to complete then extract timestamp and ↩
     variableid
  get_packet_gridstat_data[t1, t0, varid]

```

```
.reg tmp

alu[tmp, t0, AND, 0xff]
.if (tmp == 0x34 || tmp == 0x84 )
  alu[tmp, --, B, tmp]
  nop
.endif

init_rtable_offset[varid]

  // 158 instructions if RT entry is not in CAM, 103 ↩
     instructions if it is.

// extract routing info common to all output ports
// for now this means publish interval only
.reg pubint
get_stream_info[pubint]

// timestamp = timestamp + (pubint/2)
alu[tmp, --, B, pubint, >>1] ; tmp = pubint / 2
alu[t0, t0, +, tmp]
alu[t1, t1, +carry, 0]

  // 164 instructions including initial following while check.

.while ( cur_port < NUM_PORTS )
  // for all ports

  alu[--, cur_port, OR, 0]

  alu_shf[tmp, 0x1, AND, port_mask, >>indirect]

  .if ( tmp )

    dl_gridstat_filter[forward, cur_port, varid, t1, t0, ↩
       pubint]

    .if ( forward == 0 ) // whether to drop packet
      // clear bit if packet should not be sent to cur_port
      bits_clr(port_mask, cur_port, 1)

      //alu_shf[tmp, --, b, cur_port, <<1]
      //jump[tmp, port0#], targets[port0#, port1#, port2#]
      //port0#:
      //  alu[port[0], --, B, UC_NULL]
```

```
          //   br[port_end#]
          //port1#:
          //   alu[port[1], --, B, UC_NULL]
          //   br[port_end#]
          //port2#:
          //   alu[port[2], --, B, UC_NULL]
          //   //br[port_end#]
          //port_end#:

        .endif
      .endif

      rt_next_port[]
      alu[cur_port, cur_port, +, 1]

    .endw

    move[dl_next_block, BID_GRIDSTAT_NEXT1]

end_dl_gridstat_route#:

.end
#endm


#define_eval TXR_VAL0 (PACKET_TX_SCR_RING_0<<2)
#define_eval TXR_VAL1 (PACKET_TX_SCR_RING_1<<2)
#define_eval TXR_VAL2 (PACKET_TX_SCR_RING_2<<2)
#define_eval TXR_VAL3 (PACKET_TX_SCR_RING_3<<2)

#macro dl_gridstat_write_tx_req[REQ_PORT]
.begin

    .reg __tmp_a
    .if ( port[REQ_PORT] == 0 )
      // if buffer handle has the value of 0, then it's the
      // target of packet_copier

          // skip packet copier step for now
          br[end_write_tx#]
          nop
          nop

      alu[port[REQ_PORT], --, B, $rx_copy_handle]
    .endif
```

```
  alu_shf[__tmp_a, --, b, REQ_PORT, <<24]        ; 27:24 = output↩
      port
  ld_field[__tmp_a, 0111, port[REQ_PORT]]        ; 23:00 = ptr to↩
      sop buffer desc
  .io_completed scr_put/**/REQ_PORT/**/
  alu[$txreq/**/REQ_PORT/**/, __tmp_a, OR, 1, <<31] ; 31    = ↩
     set valid bit (31:28 = reserved )

  // check whether we are supposed to drop packet
  //alu[__tmp_a, --, B, REQ_PORT]
  //alu[--, __tmp_a, OR, 0]
  //alu_shf[__tmp_a, 0x1, AND, port_mask, >>indirect]
  //.if ( ! __tmp_a && port[REQ_PORT] != UC_NULL )
    // drop packet
    // TODO: support chained buffer drops
  //   dl_buf_free(port[REQ_PORT], BUF_FREE_LIST0)
  //   br[end_write_tx#]
  //.endif

  br[write_ring#]

full_ring#:
  ctx_arb[voluntary]

write_ring#:

#if ( REQ_PORT == 0 )
#define_eval SCR_STATUS SCR_RING/**/PACKET_TX_SCR_RING_0/**/↩
    _STATUS
#elif ( REQ_PORT == 1 )
#define_eval SCR_STATUS SCR_RING/**/PACKET_TX_SCR_RING_1/**/↩
    _STATUS
#elif ( REQ_PORT == 2 )
#define_eval SCR_STATUS SCR_RING/**/PACKET_TX_SCR_RING_2/**/↩
    _STATUS
#else
#error "REQ_PORT must be in {0,1,2}"
#endif
  br_inp_state[SCR_STATUS, full_ring#]
#undef SCR_STATUS

  //.set $txreq/**/REQ_PORT/**/
  //.io_completed scr_put/**/REQ_PORT/**/

  .reg zero
  alu[zero, --, B, 0]

  // test whether range of buffer handle is valid, used for ↩
     debugging
  ld_field_w_clr[__tmp_a, 0111, port[REQ_PORT]]
  .if ( __tmp_a >= 0x800 )
    nop
  .endif

  scratch[put, $txreq/**/REQ_PORT/**/, zero, TXR_VAL/**/REQ_PORT↩
     /**/, 1], ctx_swap[scr_put/**/REQ_PORT/**/]

  //alu[@count_p/**/REQ_PORT/**/, @count_p/**/REQ_PORT/**/, +, ↩
     1]
  // sig_mask |= (1<<scr_put0)

  //alu[tmp, --, B, &scr_put/**/REQ_PORT/**/]
  //alu[--, tmp, OR, 0]
  //alu_shf[sig_mask, sig_mask, OR, 1, <<indirect]

end_write_tx#:

.end
#endm


/*
 * Macro "dl_gridstat_sink":
 *
 * This macro always performs strict thread order.
 */
#macro dl_gridstat_sink[sig_mask, scr_put]
.begin
  .reg tmp

  // Force strict thread order here

  // sig_mask |= (1<<sig_prev)
  alu[tmp, --, B, &sig_prev]
  alu[--, tmp, OR, 0]
  alu_shf[sig_mask, sig_mask, OR, 1, <<indirect]

  local_csr_wr[active_ctx_wakeup_events, sig_mask]  ; tells ↩
     which signals to wait on.
  ctx_arb[--]
```

```

//.io_completed sig_copy_done
//.io_completed scr_put_copy

  // clear mask now that we waited on all signals
  alu[sig_mask, --, B, 0]


  // Because we might get a context switch before the scratch[↩
     get, ...]
  // in dl_gridstat_source, we cannot signal next thread until ↩
     after
  // scratch[get, ...] in dl_gridstat_source

  // Make sure current packet is intended for Packet TX
  alu[--, dl_next_block, -, BID_GRIDSTAT_NEXT1]
  bne[end_dl_gridstat_sink#]

  // TODO: if a packet is to be dropped, make sure to free the ↩
     corresponding buffers.
  // TODO: use port mask rather than NULL buffer handles to ↩
     determine whether to forward a packet or not

  .if ( port[0] != UC_NULL )
    dl_gridstat_write_tx_req[0]
  .endif

  .if ( port[1] != UC_NULL )
    dl_gridstat_write_tx_req[1]
  .endif

  .if ( port[2] != UC_NULL )
    dl_gridstat_write_tx_req[2]
  .endif

end_dl_gridstat_sink#:

.end

#endm /* dl_gridstat_sink */



#macro dl_gridstat_source[THREAD_ORDER, sig_mask]
.begin
```

```
  .reg tmp me
  xbuf_alloc[$rdata, 5, read]

#if (THREAD_ORDER == DL_THREAD_ORDER)
  // Wait for signal from previous thread to ensure thread ↩
     ordering
  ctx_arb[sig_prev]
#endif /* DL_THREAD_ORDER */

  // Source packet from scratch ring GRIDSTAT_RING_IN
  alu_shf[tmp, --, B, GRIDSTAT_RING_IN, <<2]
  scratch[get, $rdata0, 0, tmp, 5], sig_done[scr_get]

  // Signal that it's safe for next thread to sink and source
  br=ctx[7, thread_7#]
  signal_next_ctx[DL_SIG_WAKE]
thread_7#:


wait_for_io_signals#:

  //  We wait for the above scratch[get...] and the scratch[put↩
     ...] in dl_gridstat_sink.

  //  Add the signal to the signal mask on which to wait
  alu[tmp, --, B, &scr_get]
  alu[--, tmp, OR, 0]
  alu_shf[sig_mask, sig_mask, OR, 1, <<indirect]  ; shift 1 by ↩
     scr_get times

  // wait
  local_csr_wr[active_ctx_wakeup_events, sig_mask]  ; tells ↩
     which signals to wait on.
  ctx_arb[--]

  // clear mask now that we waited on all signals
  alu[sig_mask, --, B, 0]

  // Assembler doesn't know that we actually wait on the I/O (↩
     scratch[get..])
  // and (scratch[put..])
.io_completed scr_get
.io_completed scr_put0
.io_completed scr_put1
.io_completed scr_put2
```

```

  br!=ctx[7, thread_7_2#]
  immed[me, __UENGINE_ID]
  .if ( me == LAST_ME )
    signal_me[FIRST_ME, DL_SIG_WAKE]
  .else
    signal_next_me[DL_SIG_WAKE]
  .endif
thread_7_2#:


  // Check for empty ring
  alu[--, $rdata0, -, 0]
  beq[ring_empty#]


  // Setup meta data associated with packet

  dl_meta_init_cache[0, $rdata2, $rdata3, $rdata4, 0, 0, 0, 0]
    //dl_meta_set_nexthop_id[0xffff]  // set nexthop ID to -1

  move[dl_buf_handle, $rdata0]
  move[dl_eop_buf_handle, $rdata1]

  move[dl_next_block, BID_GRIDSTAT]

  // Exit macro
  br[end_dl_gridstat_source#]

ring_empty#:

  //alu[@count_empty, @count_empty, +, 1]

  // Output NULL Packet
  move[dl_buf_handle, IX_NULL]
  move[dl_next_block, IX_NULL]

end_dl_gridstat_source#:

  xbuf_free($rdata)

.end
#endm /* dl_gridstat_source */


```

```

#macro dl_gridstat_init[]
.begin

  .reg tmp me

  immed[me, __UENGINE_ID]


    //  Context common initializations

    // get context num
    //local_csr_rd[active_ctx_sts]
    //immed[ctx_num,0]  //immed operand contains local_csr_rd ↩
       data
    //alu[ctx_num, ctx_num, AND, 0x07]

    move[sig_mask, 0]
    cam_clear


    //  Context specific initializations

    .if( ctx() == 0 )
      //alu[@count_p0, --, b, 0]
      //alu[@count_p1, --, b, 0]
      //alu[@count_p2, --, b, 0]
      //alu[@count_p3, --, b, 0]
      //alu[@count_empty, --, b, 0]
      //alu[@debug_cnt, --, b, 0]

      .reg a1 a0 rec n d rem

      immed32(a1,  0x00000039)
      immed32(a0,  0x000a0007)
      immed32(n, 4)
      immed32(d, 30)
      immed32(rec, 0x88888888)
      remainder64x32[rem, a1, a0, d, rec, n]

    .endif


    //  All threads wait for system go signal

    .if( ctx() == 0 )

      // this is the signal that all blocks are to
      // wait on to indicate that system initialization
      // is done

      ctx_arb[system_init_done]

      // signal all other contexts to proceed

      alu[tmp, --, B, &sig_prev, <<3]
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 1
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 2
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 3
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 4
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 5
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 6
      alu[tmp, tmp, +, 1]
      local_csr_wr[SAME_ME_SIGNAL, tmp] //thread 7
      alu[tmp, tmp, +, 1]

    .else

      .set_sig sig_prev
      ctx_arb[sig_prev]

    .endif

    // kick start next thread sequencing by thread 0 of the ↩
       first ME signaling itself

    .if( ctx() == 0 && me == FIRST_ME )

      alu[tmp, --, b, &sig_prev, <<3]
      local_csr_wr[SAME_ME_SIGNAL, tmp]

    .endif

.end
```

```
#endm /* dl_gridstat_init */


/*
 *
 * GRIDSTAT DISPATCH LOOP
 *
 * This is where the microengine start executing code.
 *
 */

// Initialize
dl_gridstat_init[]

// Get packet from Packet RX
dl_gridstat_source[DL_THREAD_ORDER, sig_mask]

// instruction count 0 here.

loop#:
    // Compute meta data associated with this packet.
    dl_gridstat_route[sig_mask]; // 239 instructions + 88 * ↩
        RTcomplexity.

    // Send packet to Packet TX
    dl_gridstat_sink[sig_mask, scr_put]; typically 50 ↩
        instructions.

    // Get next packet from Packet RX.
    // We can call the source macro with no thread order here ↩
        because
    // strict thread order is already enforced in ↩
        dl_gridstat_sink macro
    // called immediately before source.
    dl_gridstat_source[DL_THREAD_NO_ORDER, sig_mask]; 30 ↩
        instructions.

br[loop#]

// makes a total of 320 + 88 * RTcomplexity instructions.
// e.g. RTcomplexity of 2 makes 496 instructions.
```