FAST SIMULATION OF PROGRAMMABLE NETWORK FORWARDING PLANE DEVICES

SHAFIGH PARSAZAD

A Thesis
in
The Department
Of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements for the
Degree of Master of Applied Science (Electrical and Computer Engineering)
at

Concordia University
Montréal, Québec, Canada

July 2016

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Shafigh Parsazad

Entitled: "Fast Simulation of Programmable Network Forwarding Plane Devices"

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

Complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Yousef R. Shayan

_____ Examiner
Dr. Benjamin Fung

_____ Examiner
Dr. Ferhat Khendek

_____ Supervisor
Dr. Samar Abdi

Approved by: _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

July 2016

_____
Dr. Amir Asif, Dean
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

Fast Simulation of Programmable Network Forwarding Plane Devices

Shafigh Parsazad

With the evolution of the Internet, the processing of packets at the routers while providing flexibility in deploying new protocols and services at the same time has become a major concern. Programmable forwarding elements with high processing capability have emerged as a solution. But the main challenge is to find the optimal hardware architecture while taking into account constraints such as different packet processing functions, task scheduling options, electrical power consumption and providing quality-of-service (QoS) guarantees. Therefore, it is essential to investigate methods that help in identifying limitations and bottlenecks before physical fabrication. Having an appropriate model provides designers a progressive path to narrow the design space and establish credible and feasible alternatives before deciding on an implementation.

In this thesis, we propose a flexible and fast instruction accurate host-compiled simulator to make it possible to explore wide ranges of architectures and application scenarios to find the optimal configuration that meets given performance, throughput and latency for programmable forwarding elements. Application developers can use the simulator as a virtual prototype to simulate and debug their applications before hardware availability. Moreover, forwarding device architects can use simulator to evaluate the trade-offs between different hardware/software design decisions.

ACKNOWLEDGEMENTS

I would like to take this opportunity to sincerely thank my advisor, Dr. Samar Abdi who has given me much thoughtful academic advice and has always been on my side unconditionally. Without his insightful guidance, inspiration and constructive feedback this work would not have been achievable.

I am also thankful to my colleagues in Embedded Systems Lab including Ehsan Saboori, Umair Aftab, Faras Dewal, Kamil Saigol, Gordon Bailey, Eric Tremblay, Aryan Yaghoubian and Sara Karimi for their great company and all their help.

Most of all, I would like to thank my wonderful wife Azin, for being my best friend and whose love, and understanding I will cherish forever. Last but not least, I am truly thankful to my parents for supporting me throughout all my studies. Words cannot express my appreciation for the countless sacrifices they made.

Contents

List of Figures VIII
List of Tables X
Code Listings XI
List of Acronyms XII
Chapter 1: Introduction 1
1.1. Motivation 1
1.2. Related Work 3
1.2.1. Cycle Accurate Simulation 3
1.2.2. Functional Simulation 5
1.3. Methodology 7
1.4. Main Challenges and Contributions 8
1.4.1. SystemC Model Semantics 8
1.4.2. Timing Estimation with Dynamic Instruction Count 12
1.4.3. Timing Estimation with Instruction Set Simulator 13
Chapter 2: Programmable Forwarding Platform 15
2.1. Programmable Forwarding Elements 15
2.1.1. ASIC 17
2.1.2. CPU (Soft Switches) 18
2.1.3. Many Cores Network Processors 19
2.1.4. Programmable Pipeline 22
2.2. Quality Metrics 23
2.2.1. End-to-end Packet Delay 23
2.2.2. Throughput 24
2.2.3. Packet Loss 24
Chapter 3: NPU SystemC Model Semantics 25
3.1. Overview 25
3.1. Basic Workflow 26
3.2. NPU Modules 27
3.3. On-Chip Network 28
3.3.1. Routing Algorithms 29

3.3.2. Router Modeling 29
3.4. Observers 30
3.5. Functional Validation 33
3.6. Limitations 35
Chapter 4: Timing Estimation with Dynamic Instruction Count 37
4.1. Host Compiled Simulation 38
4.2. Dynamic Instruction Count Pass 38
4.3. Simulation Results 41
4.3.1. Average end-to-end Packet Delay 41
4.3.2. Traffic Management Policies 42
4.3.3. Packet Loss 45
4.3.4. Power Consumption 46
4.3.5. Simulation Speed 47
Chapter 5: Timing Estimation with Instruction Set Simulator 49
5.1. Overview 49
5.2. Instruction Set Simulator 50
5.3. SystemC/OVP Simulation 51
5.3.1. OpenRISC 51
5.3.2. Open Virtual Platform (OVP) 52
5.3.3. Simulation Results 56
5.4. Back-annotation using ISS 62
5.4.1. Direct Annotating 62
5.4.2. Corrective Feedback Annotation 63
5.4.3. Simulation Results 64
5.5. Timing Estimation Summary 67
Chapter 6: Conclusion and Future Work 68
6.1. Conclusion 68
6.2. Future Work 69
6.2.1. Accurate Power Consumption Estimation 69
6.2.2. Back-annotation Using Prediction 71
6.2.3. Automated Design Space Exploration Framework 71
Appendix 72
1. Observers 72

2. Timing Estimation with ISS integration 76
Bibliography 82

List of Figures

Figure 1. Forwarding Element Architecture 16

Figure 2. Typical network processor architecture 20

Figure 3. Router modeling in SystemC 30

Figure 4. Example observers developed 31

Figure 5. Thread safe observers 32

Figure 6. Observers 33

Figure 7. Model Validation 34

Figure 8. Design evaluation based on timing information 37

Figure 9. Dynamic Instruction Count Pass 40

Figure 10. Application intermediate code before and after applying DICP 40

Figure 11. Average end-end delay 42

Figure 12. Scheduling policies 44

Figure 13. Number of drops versus Input rate 46

Figure 14. Average idle time versus Input rate 47

Figure 15. Integration of SystemC model with OVP CPU model 55

Figure 16. Longest Prefix Match 57

Figure 17. Connection modeling between clients and servers 58

Figure 18. Simulated instruction/ forwarding table entries 60

Figure 19. OVP increasing simulation time as input load increases 61

Figure 20. Workflow of experiments..... 63

Figure 21. Corrective feedback annotation to tune parameters 64

Figure 22. Proposed back-annotation method evaluation..... 65

Figure 23. Proposed back-annotation method memory usage..... 66

List of Tables

Table 1. LLVM back-annotation tool speedup 48

Table 2. SystemC/OVP Simulation evaluation 60

Code Listings

Listing 1. High level of abstraction in SystemC 9

Listing 2. XY routing protocol 29

Listing 3. Read callback function 54

List of Acronyms

CA	Cycle Accurate
CPU	Central Processing Unit
DICP	Dynamic Instruction Count Pass
DSE	Design Space Exploration
IA	Instruction Accurate
ISS	Instruction Set Simulator
LLVM	Low Level Virtual Machine
NoC	Network on Chip
NPU	Network Processing Unit
OVP	Open Virtual Platform
P4	Programming Protocol-Independent Packet Processors
PE	Processing Element
QoS	Quality of Service
RMT	Reconfigurable Match Table
SDN	Software Defined Networking
SoC	System on Chip
TLM	Transaction Level Modeling

Chapter 1: Introduction

1.1. Motivation

In today's network, data rates are increasing, protocols are becoming more dynamic and sophisticated and traffic for video and data applications is expected to grow exponentially. This is mainly driven by the popularity of video sharing and streaming applications (e.g. Netflix and YouTube) and smart mobile devices, tablets will certainly make this a reality. The switches, routers, and other devices within these networks have become exceedingly complex because they implement an ever-increasing number of standardized distributed protocols and proprietary interfaces. These challenges cannot be properly addressed with the rigid solutions provided by today's networking equipment. Networks need the ability to respond to changes, faults and scale performance to handle large volumes of client requests without creating unwanted delay.

To capitalize on this condition, telecom operators need to have more flexible, scalable and energy efficient processors, specifically designed for packet processing. One of the primary

challenges in the design of network components, such as network processing units is to determine the best hardware architecture to support diverse applications. We need to quickly adapt to new conditions and prevent network from becoming congested. Designing such architectures is partly complicated by the fact that they involve complex trade-offs between flexibility and efficiency for today's evolving network elements, flexibility to adapt to new functional requirements and ability to provide scalable performance in response to increasing line rates[1].

The idea of programmable networks has recently re-gained considerable movement due to the emergence of the Software-Defined Networking (SDN) paradigm. SDN, Through promoting (logical) centralization of network control and network programmability, promises to dramatically simplify network management and enable innovation[2]. But, there are many questions to be answered. What are the various cases that need to be supported? What type of programmatic interfaces should be presented? In order to answer these questions, new services, applications and protocols should first be developed and tested on an emulation of the anticipated deployment environment before moving to the actual hardware. To evaluate design options and guarantee correct early design decisions, modelling techniques require the capability to evaluate different architectures effectively and still accurate enough in early design phases.

The design space for forwarding elements (e.g., number of processors, interconnections, scheduling options, etc.) can be very large due to the diverse workload, application requirements, and system characteristics. Therefore, it is essential to investigate methods that help in identifying limitations and bottlenecks in network processor implementation before physical fabrication. Having an appropriate model provides designers a progressive path to narrow the

design space and establish credible and feasible alternatives before deciding on an implementation.

In this thesis, we propose a flexible accurate host-compiled simulator (the term host is used for system that runs the simulator and the term target for the system being simulated) to make it possible to explore wide ranges of architectures and application scenarios to find the optimal configuration that meets given performance, throughput and latency for programmable forwarding elements. It relies on the back-annotating the host compile simulator from timing characteristics obtained an instruction set simulators (ISSs), Open Virtual Platform.

The aim of this work is to provide designers with the possibility of faster and efficient architecture exploration at a higher level of abstractions, starting from an algorithmic description to implementation details.

1.2. Related Work

Simulators have been used by many computer architect researches to validate new proposals, as designing new hardware requires significant investments in both time and money. Many simulators have been developed to solve different research challenges and they vary in the details they model. In this section, we introduce different kinds of architectural simulators for different types of evaluation.

1.2.1. Cycle Accurate Simulation

Models of different level of details for various purposes are proposed to construct a proper system platform for accuracy and performance trade-off. To eliminate pins and wire details, and to improve simulation speed while not losing cycle timing accuracy, cycle-accurate (CA) models are proposed. Cycle accurate models are used for synthesis and are fairly close in structure to

RTL level models[3] . Due to the excessive level of details and enormous number of simulated states that is captured at this modeling abstraction, the simulation speed is quite slow, but still anywhere between 10 and 100 times faster than RLT simulation, depending on the complexity of the design[4]. Besides, enormous time that it takes to create these models, which is also comparable to the time takes for writing details RTL code, makes using these models quite problematic, both in their modeling time and slow simulation speed to be useful for high level architectural exploration of SoC design today. Regardless of these drawbacks there have been a few approaches that have used cycle accurate models for network processors.

For the x86 platform, numerous functional simulators have been developed. Bochs[5] is a well-known open source x86 simulator, with support for nearly all x86 features. However, Bochs is very slow and is not useful for implementing cycle accurate models of modern out of order x86 processors (i.e. it does not model caches, branch prediction, etc.). PTLSim[6] is a cycle accurate full system x86 athlon microarchitectural simulator. PTLsim models a modern superscalar out of order x86-64 processor core and unlike Boche models execution at a level below the granularity of x86 instructions.

Intel's performance model framework: Asim[7] and AMD's simulator[8] are fastest true cycle-accurate x86 simulators, which run at 1KHz to 10KHz, requires up to ten years to simulate a 3GHz target for 2 minutes[9]. At such speeds, it is still impractical to use real program runs to explore, evaluate and refine microarchitectures, considering simulators are getting slower as the number of target cores and their complexity increase. MCSimA+[10], introduced a simulator of complex x86 cores, which takes advantage of full-system simulators such as gem5 [11] (full-system mode), Simflex[12] and application-level simulators: SimpleScalar[13] and Graphite[14].

While these simulators are useful to model general purpose processors simulators at cycle level, but they are not capable of reflecting the detailed architecture of packet forwarding elements such as network processor units or programmable pipelines.

NePSim[15] is a cycle-accurate simulator of the Intel IXP1200 network processor. The simulator is for performance and power consumption evaluation of network processors. Although NePSim achieves satisfactory accuracy but it is not configurable and only matches a specific Intel network processor: IXP1200, which makes it unsuitable to adapt to new specification changes.

1.2.2. Functional Simulation

When looking at more realistic SoC implementations of today, the architectural assumptions of the early works have changed. First, a system may contain not only one, but several CPUs of different types such as reduced instruction set computer (RISC) cores, digital signal processors (DSPs), application specific instruction set processors (ASIPs), or very long instruction word (VLIW) processors. As architectures and systems become more complex, conducting a single, cycle-accurate simulation experiment can take from days to weeks[16].

In general, cycle accurate simulators are several orders of magnitude slower that the systems they simulate[17]. on a 2.4GHz Xeon workstation SimpleScalar out-of-order timing simulator can simulate a set of SPEC CPU2000 benchmarks [18] at a speed of about 300K instructions per second. This speed difference leads to two problems: First, slow simulation rate forces simulators to be usually limited to the first few billion instructions, which reflects only less than 10% the execution time of many standard benchmarks. Since such simulation studies only cover a small part of the applications, they face the risk of reporting unrepresentative

behavior. Second, excessive level of details in cycle accurate simulators prevents them to be suitable for hardware/software co-design studies where rapid turn-around is necessary. Hardware/software codesign, sometimes also named software/hardware codesign or just codesign equivalently, started to be considered as “the process of concurrent and coordinated design of an electronic system comprising hardware as well as software components based on a system description that is implementation independent by the aid of design automation”[19].

To reduce the time per experiment, as stated earlier, architects may choose to reduce workloads[20], or to run detailed simulation only on pieces of an application. Performing detailed, cycle-accurate simulation on only selected parts of the system dramatically reduces the time to conduct a simulation experiment. Unfortunately, functionally simulating program behavior up to the portion(s) of interest which to begin cycle-accurate modeling can still take a considerable amount of time. In fact, according to [21], execution times of such hybrid simulations are in many cases still dominated by the functional simulation time , which can be significant for SPEC 2000 codes.

To cope with today’s increasing complexity of SoC systems and support design space exploration, it is natural to expect that simulation techniques should evolve to operate at higher levels of abstraction, where they could exploit inherent advantages such as an increase in simulation efficiency. However at higher levels of abstraction, it is possible to lose details.

One problem network researchers often face is how to test new network topologies and protocols in a quick, inexpensive, and realistic manner. As many researchers can attest, building test beds costs valuable time and money, with the added issue that test beds are limited in size and thereby do not represent realistic networks[22]. Mininet[23] was developed to solve these issues. Mininet is a network emulator which leverages Linux features such as virtual Ethernet

pairs and separate network namespaces to emulate many-host networks. Mininet is a network emulator that allows the user to easily create, customize, share and test SDN networks, yet it does not scale to large networks[24]. Fs-SDN is another tool for prototyping and evaluating new SDN-based applications. Although both Mininet and Fs-SDN aim to scale to network topologies comparable to the size of a modern data center, but they do not model the forwarding plane architecture and are therefore unsuitable for making architectural design decisions.

1.3. Methodology

A forwarding element processor is a highly integrated complex system which can have more than 100 cores, and if we instantiate detailed model for each core, we'll have huge slowdown in our simulation. The goal of our work is to figure out how to model these cores as accurately as possible and how to make sure we do not sacrifice simulation speed. While simulators at high-level of abstraction may not be appropriate for conducting micro architectural research on forwarding element processors, low-level detailed system simulators usually are relatively slow. In general, the more detailed architecture the simulator can handle, the slower the simulator simulation speed. A model is then defined by the amount of implementation detail, i.e. by the amount of target-specific computation and communication layers explicitly included. Any implementation layers below a certain interface are abstracted away and replaced by an abstract model of the underlying target functionality and timing.

Our Transaction Level Modeling (TLM) approach[25] is different from the above mention methods in that, it takes advantage of a high level abstraction simulator where details of communication among modules are disengaged from the details of the implementation of the functional units. In other words, by focusing more on the functionality of the data transfers and

less on their actual implementation, system-level designers can effectively evaluate different architecture alternatives.

By using the transaction-level modeling abstraction in SystemC, a system can be specified at various levels of abstraction. Whenever increased level of abstraction is beneficial, such as high speed co-simulation of hardware and embedded software, models of different parts of the system can co-exist during system simulation at different levels of abstraction. Through the suppression of "uninteresting" details, TLM enables higher simulation speed than cycle-accurate modeling. The software part of a system can be naturally described in C or C++. Interfaces between software and hardware and between hardware blocks can be easily described either at the transaction-accurate level or at the cycle accurate level. Typically transaction-level models are used for functional modeling (both timed and untimed), platform modeling and for constructing testbenches[26].

1.4. Main Challenges and Contributions

The main contributions of this thesis are presented as follows:

1.4.1. SystemC Model Semantics

In the past, the hardware design world standardized on two languages: VHDL and Verilog, but as SoCs have grown more complex, with multi-million gate designs, and the increasing pressure to get designs out faster with first-time design success, rising productivity is vital to design modern electronic systems. Without productivity advances, many new system concepts will be impractical. To help manage this, engineering teams are experiencing great efficiency by starting their tasks with a high-level C++ model to verify functionality[27]. SystemC is a C++ class library which provides an event-driven simulation kernel in C++, mimicking hardware

description languages VHDL and Verilog. While such languages are often used for Register Transfer Level descriptions, SystemC is generally applied to system-level modelling, application development and evaluating architectural tradeoffs, while the system is still in its design phase.

In this thesis we take the best of both high-level and cycle-accurate simulation and propose a new approach based on Transaction Level Modeling with SystemC. The goal is to provide the speed and ease of development of a high-level simulator, while being able to reflect the forwarding plane architecture by using different levels of abstraction for the accuracy. By leveraging C++, SystemC efficiently supports object-oriented capabilities, such as templates, classes, polymorphism, and operator overloading, can be used to manage design complexity in hardware in the same way as in software. For example, when reading a memory at high level of abstraction, source code can be as simple as follows:

```
1: int address;
2: int value;
3: int memory[1024];
4: …
5: value = memory[address];
```

Listing 1. High level of abstraction in SystemC

By abstracting the hardware access to a memory, using an array access notation as shown above, or hardware services, SystemC offers engineers to concentrate on the actual functionality of the system more than on its implementation details. The details are abstracted, making the code compact and readable. Those details are instead implemented in a C++ class that uses SystemC to set the chip enable and read/write signals and to drive the address signals, etc.

A. Functional Validation

Simulators are primarily used as a vehicle for demonstrating or comparing the utility of new architectural features, compilation techniques, design space exploration and early-stage

feasibility, rather than performance validation of an actual chip design. As a result, rather than implementation details, or comparing with actual chip absolute metrics like timing, it is more important to validate functionality of the system under design that is indifferent to vendor-specific physical devices and be able to compare different designs metrics with each other to find the optimal and feasible ones.

In order to investigate functional validation of the simulator, different types of applications will be executed on different architectures of forwarding elements. To test the functional correctness of forwarding elements implementation themselves, application output (including numerical results) was compared with the output from soft-switches. The soft-switch was modeled as host-compiled C code inside a SystemC wrapper, so that it can be simulated with our traffic pattern. The soft-switch does not have any timing and no underlying platform model. As such, it is useful for functional validation of the application. Under this technique, programs are viewed as formal objects developed from a set of precise specifications. Once developed, they are guaranteed to produce the output given in the specifications. Developing a program requires several separate activities: (1) designing a specification that expresses the task to be performed, (2) refining that specification into a formal explicit statement that captures the specification's intended functionality, and (3) developing a program that correctly implements that functionality.

B. Speed

As discussed already, design teams need simulators throughout all phases of the design cycle. It might also be noted that simulation is not cheap. For full system designs, runs of 20 days (24 hours a day) on a single user multi-million instruction per second machine have been reported[28], and as systems become more complex, it is expected that time to complete the

simulation get worse rather than better. This emphasis the need for considering the speed of any proposed simulator. One of the advantages of using SystemC is its high execution speed, due not only to its simulation engine itself, but by the high abstraction level generally used for SystemC based system description. One of the interesting research areas would be to see the trade-off in simulation speed between a model that captures architectural details of a forwarding element compared to a pure software switch.

C. Observers and Design Space Exploration

The inability to change the hardware, once built, and time to market constraint, increases the cost of any fault in the final chip design. The relative cost of finding faults at different stage of the production, namely: design time, chip-test time, board construction time or finished silicon board is estimated to be 1:10:100:1000[28]. However, any software running on a computer is relatively easily modified. Domain-specific measurements can be used when simulating particular architectures in terms interesting to the user, to analyze and characterize various factors that contribute to the design performance and examining the effect they have. For example network application users might be interested in metrics like throughput, average packet latency, packet loss or power estimation. Such measurements require some custom tool that is able to observe the simulation output and interpret it in domain specific terms. They allow the comparison of various architectures and find the optimal ones regardless of any particular slowdown of the simulation. Observers can also be used to hook-up between the SystemC simulation and visualization of results in MATLAB.

1.4.2. Timing Estimation with Dynamic Instruction Count

Estimation of timing plays an important role in making architectural design decisions. For instance, end-to-end delay is considered as an important parameter to analyze the performance of the network. The end-to-end packet delay is the time it takes to deliver a packet from ingress to egress. Without timing estimation, no end-to-end delay comparison between different architectures can be done. Of course, since SystemC is a simulation language, the simulated time is different from the wall-clock time. The intrinsic behavior of the SystemC discrete-event simulation engine is such that the simulation produces a sequence of simulation instants. Computations occur only at these simulation instants: the simulation intervals only correspond to the increment of simulated time in the scheduler[29]. Timing estimation simulates execution of the application on target platform. In the first approach we used DICP for timing estimation, which is our custom built tool which is used to analyze application byte-code at Basic Block (BB) level. The following challenges should be addressed for timing estimation.

A. Processing Elements

Timing annotation can be defined by user for fixed function processing elements like parser or scheduler. This timing can be annotated at source level in threads inside each one of these processing elements for function calls, statements, etc. Modeling an action that takes time (say, $load\_img()$, taking 3ms) is not directly possible in SystemC, which can express only instantaneous computations. The duration can be modeled with a $wait(time)$ statement. A common practice is to run the functional behavior first, followed by the wait statement (e.g. $load\_img()$; $wait(3, SC\_MS)$;). Duration of the action being performed can also be computed inside the function $load\_img$ as t during execution of the task, then the following wait

can be performed as $wait(t)$ to simulate execution of the function on the target. These timing values can also be derived from measurement or low-level simulations.

B. Application Timing

The C/C++ code targeted for the NPU cores is either provided by the user or generated from P4[1]. The code is wrapped inside a SystemC thread (SC_THREAD), and instantiated inside the SC_MODULE corresponding to the core. Timing annotation in the application and hardware model simulates the execution on target platform. The timing model for this code may also be provided by the user by adding SystemC wait statements at the source level, although this is both cumbersome and impractical.

If the application code is targeted for the NPU cores, we provide a timing annotation utility that automatically inserts basic-block level timing in the application source. The timed application is compiled along with the SystemC model of the NPU and the module logic to generate the final simulation binary.

1.4.3. Timing Estimation with Instruction Set Simulator

In general, host-compiled simulators can provide significant speedups (reaching simulation speeds of several hundred MIPS), but often focus on functionality and speed at the expense of limited or no timing accuracy.

In order to offer a better trade-off between accuracy and speed of performance estimations, the framework supports back annotating host compiled simulation from an integrated OVP framework inside SystemC. Instruction Set Simulators (ISS) functionally behave as model CPU, but they do not map any internal architecture and thus – they simulate four to five rows faster,

[1] P4 is a high-level declarative language for programming protocol-independent packet processors..

compared to the RTL. Since ISS is intended to control rest of the SoC, it must be equipped with appropriate interfaces to communicate other components. As the solution, TLM standard was chosen to be implemented alongside with ISS modules. Transaction-Level Modeling (TLM) has become tremendously popular and almost universally accepted as a vehicle for acceleration of platform model integration by having different level of abstractions inside the model. The resulting approach offers faster and easier software development while enabling accurate analysis thanks to the integrated instruction-accurate simulator.

Chapter 2: Programmable Forwarding Platform

2.1. Programmable Forwarding Elements

The aim of this section is to give an overview on what happens to a packet inside a switch or router, regardless of the architecture by presenting a typical router design, but the same principle can be applied to forwarding elements. Then, different technologies to build a switch/router namely Application-Specific Integrated Circuits (ASICs), Network Processing Units (NPUs), Software switches and Reconfigurable Match-Actions switches (RMT) will be covered.

The Fundamental task of any router as illustrated in Figure 1 is to switch a packet from an input link to the appropriate output link based on the destination address in the packet.

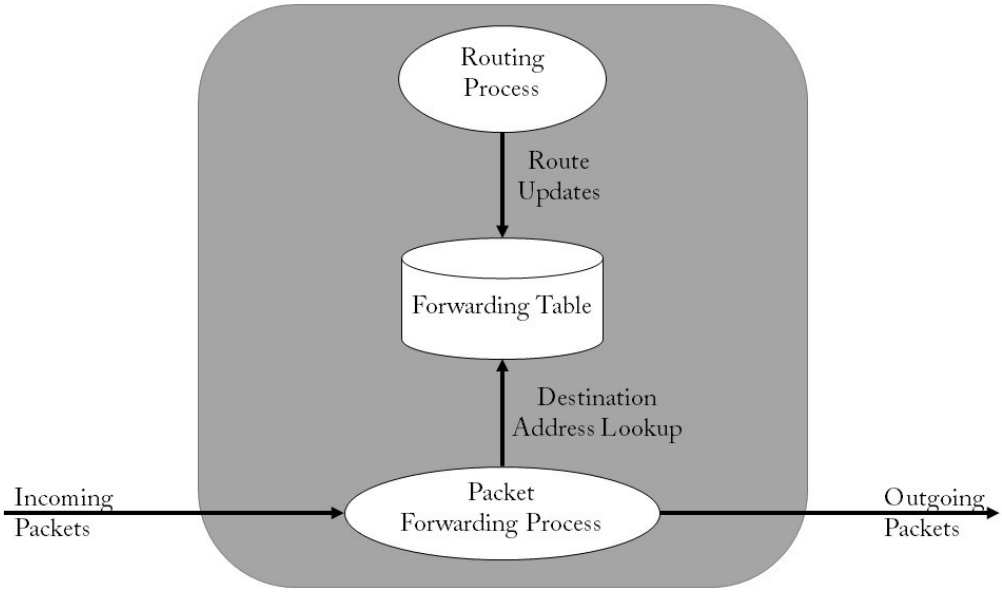

Figure 1. Forwarding Element Architecture

Any routing system has to have four essential components to implement the routing and packet-forwarding process: routing procedure, packet forwarding processing, a switch fabric and line cards. Routing procedure performs the routing function and is responsible to run the routing protocols, creating forwarding tables which is accessed by the packet processing part of the system.

When a packet first enters the router, it is received by input port (ingress). Then regardless of a router's architecture, each packet entering the system requires certain amount of work such as IP header validation, checksum recalculation and most importantly route lookup, which is searching the forwarding table for determining the output port for the packet. When the forwarding decision for the packet is made, it can be moved to the switch fabric. The switch fabric does the actual transfer of the packet from an input to an output port. It can be designed

using a few different methods depending on the performance requirements. Finally after having passed the switch fabric the packet arrives at the output port (egress).

Technically speaking routers and switches work in a similar fashion. The difference lies in the information that is used to do the forwarding decision. Routers use Internet Protocol (IP) addresses, and thus operate on layer 3, for the forwarding decisions while switches use Media Access Control (MAC) addresses, and thus operate on layer 2 only. This difference leads to routers needing to run different routing protocols to gather information about the network and to construct routing and forwarding tables[30]. Typically switches are used in smaller networks where less control over the data flows is needed, whereas routers are used in big installation where control is important.

2.1.1. ASIC

Perhaps the common approach when designing high-performance routers to do a predefined set of tasks is using application-specific integrated circuits (ASICs) coupled with specialized, high speed memory (e.g. , TCAM) to store forwarding tables. The common requirement for the typical ASIC task is that it needs to be done at line rate for the incoming packets. Typically ASICs are used when such performance is needed that it cannot be achieved with regular Central Processing Units (CPUs) nor NPUs (Network Processor Units). Arriving packets are processed by a fast sequence of pipeline stages, each dedicated to a fixed function. These devices can be very efficient for the set of functions, but there are some drawbacks to using ASICs though. While these chips have adjustable parameters, they fundamentally cannot be reprogrammed, which makes adding new features like recognizing or modifying new header fields a very demanding task. In the worst case scenario a customer that wants to utilize this upgrade needs to buy a whole new piece of equipment instead of just upgrading software like one would do with

the alternatives that offer some sort of programmability[30]. In other words, inability to modify the hardware forwarding algorithm without respinning the chip, forces packet forwarding to evolve on hardware design timescale which are extremely slower compared to the rate at which network requirements are changing. Each time a new feature needs to be added, the production of the old ASIC needs to be stopped and add an extension to the original chip or, a totally new chip, designed, which can take up to 2-3 years to support new features. Moreover, because ASICs are not programmable we might waste valuable resources of the switch if it is not used in the network.

2.1.2. CPU (Soft Switches)

Typically, software switches run a software on a CPU (general purposed processors, commodity hardware) like x86. Software-based switches on commodity hardware can affordably store large tables in SRAM (CPU cache) or DRAM. The typical software router has a motherboard that is used to connect the CPU/CPUs, NICs (Network Interface Cards) and other components together.

Software routers and switches have the potential to become a solution that is used when a cost effective method for packet forwarding is needed. What they do not offer is the huge performance that is required in backbone networks. On the other hand, because everything that the software router does is done in software, new protocols and other required changes can be easily made by altering the source code. Most software routers run on some form of Linux operating system, which further increases the ease of making changes as anyone can access the code freely.

The journey of a packet through a Linux software router starts at one of the RSS receive queues from where it is transported, via software interrupt and NAPI up the network stack and

into the switching function that decides which port the packet should be output to. From here on the packet then goes to an output queue from where it is transported by software interrupt to the NIC and onto the wire.

Another example of a software switch is cuckooswitch[31]. Conventional hash-table based lookups are typically memory-inefficient, which becomes important when attempting to achieve large scale or fit lookup tables into fast, expensive SRAM. CUCKOOSWITCH combines a new hash table design together with Intel's packet processing architecture (the Intel Data Plane Development Kit[32], or DPDK for short) to create a software switch.

2.1.3. Many Cores Network Processors

Network processors or NPUs form a family of highly specialized programmable processors. They are mainly used to optimize packet processing functions in routers, switches and other networking equipment. When designing network equipment, choosing flexibility or performance leads to the choice of using a general purpose CPU, an NPU or a fixed function ASIC. ISPs require solutions that scale in terms of both routing richness and packet-forwarding performance. Network processing units offer a solution that lies somewhere between the programmability of a CPU and the performance of an ASIC. Like a CPU, NPUs are programmable. But NPU's architecture is optimized to handle data networking and support software based implementations of the critical paths while processing packets at high speeds. Network processors can be categorized in many ways[30]. One way of categorizing NPUs is by their throughput performance to entry-level (1-2 Gbps), mid-level (2-5 Gbps) as metro NPUs and high-end (10-100 Gbps) NPUs as Core NPUs[33]. Core NPUs run faster to handle applications such as high-end routers used by carriers that serve large numbers of customers. Metro NPUs are for lower-speed applications, such as office switches.

Generally, NPUs consist of the following basic components: several network interfaces to the attached networks, processing module(s), buffering module(s), and an internal interconnection unit (or switch fabric). Typically, packets are received by an inbound network interface, processed by the processing module and, possibly, stored in the buffering module. Then, they are forwarded through the internal interconnection unit to the outbound interface that transmits them on the next hop on the journey to their final destination.

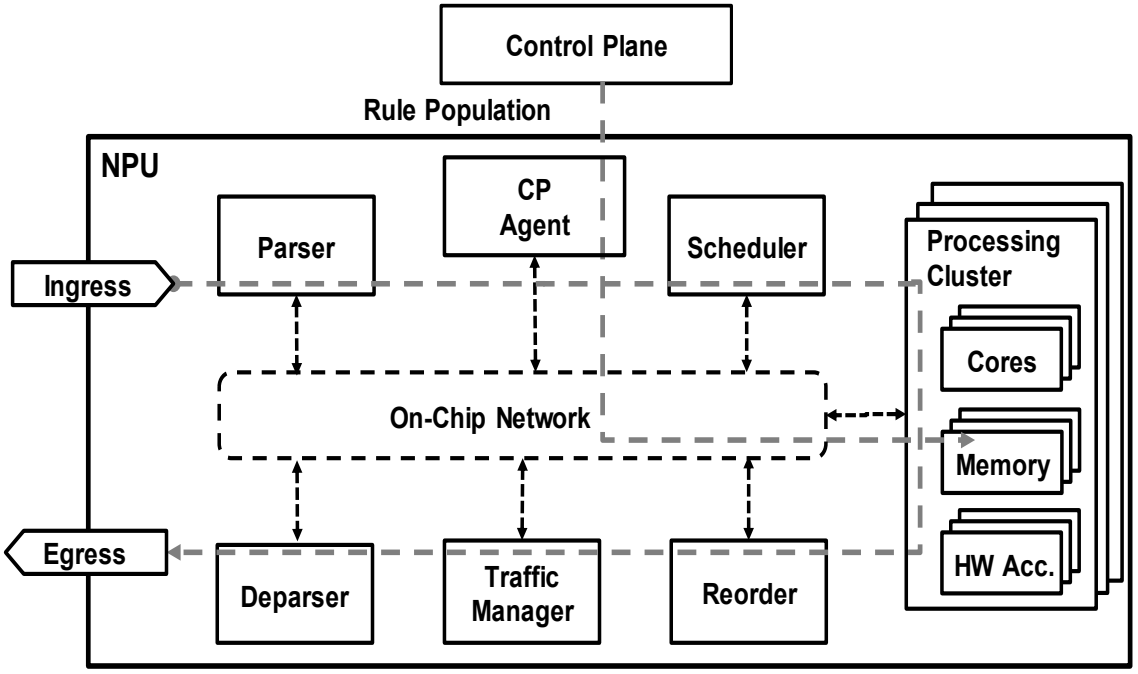

Figure 2. Typical network processor architecture

Figure 2 illustrates the high-level architecture of a simplified NPU, inspired by the SNP 4000 architecture[34]. The NPU consists of configurable hardware units for parsing, scheduling, reordering, traffic management and deparsing of packets. It consists of multiple identical processing clusters, each consisting of a set of RISC cores, memories and hardware accelerators. The control plane populates the match table entries in the NPU's memory via an agent. Packets arriving at the ingress are sent to the parser for classification and generation of a packet

descriptor. The scheduler then assigns the packet to one of the processing clusters. The packet processing application runs as concurrent identical threads on the CPU cores in the clusters, in a run-to-complete fashion. The application thread transforms the packet header, inspects the associated payload if needed, and then waits for the scheduler to send the next packet. Since any two packets may be processed on different cores, they may encounter different delays and may leave the processing clusters in a different order than the one they arrived. Therefore, a reorder module is used to restore the ordering of packets. The packet then goes to a traffic manager that sends the packets out according to their priority. The deparser generates the deparsed header, and recombines the header with its payload before sending the packet out on egress.

The on-chip communication architecture of an NPU may vary from design to design. However, the large number of processing cores and the requirement to access the same set of modules from several cores, naturally lends itself to a network-on-chip architecture.

The bulk of the packet processing on an NPU takes place in the processing cluster, and the majority of the packet latency results from the search-and-lookup operations in memory. Therefore, NPU designers can benefit from executable models that enable them to evaluate the trade-off between the number of cores and the budgeting of on-chip memory in their design. Such an executable model can also help with fast and early functional validation of the NPU architecture before hardware availability.

The programmability of the NPUs enables new features to be added to the equipment after it has been deployed. Although, one of the challenges is the differences in programming the NPUs. One model hides programming details, which simplifies the process but gives programmers less control. The other gives programmers access to many details, such as those needed to optimize code to process data streams across numerous processors. These architectural

choices lead to difficulties in estimating the software development costs and to longer than expected software development projects.

2.1.4. Programmable Pipeline

The functionality of the Reconfigurable Match Table (RMT) switches are based on the approach known as “Match-Action”. A small number of fixed fields in the packet for classification is matched against a table, while every match is paired with a corresponding action(s) that are applied to the packet. One can imagine implementing Match-Action in software on a general purpose CPU. But for the speeds that is required to keep up with today’s networks—about 1 Tb/s—the parallelism of dedicated hardware is a must. Switching chips have remained two orders of magnitude faster at switching than CPUs for a decade, and an order of magnitude faster than network processors, and the trend is unlikely to change[35]. RMT switches are implementing Match-Action in hardware to exploit pipelining and parallelism, while having enough reconfiguration, so that new types of packet processing could be supported at run-time.

In general, RMT switches consist of a parser to enable matching on fields, followed by a set of pipeline stages, each with a match table of arbitrary depth and width that matches on fields. A packet header vector, which is a set of header fields, can be created through parser. Parser should allow any modification for the packet header, such as defining new fields or adding fields, implying a reconfigurable parser. The vector flows through a sequence of logical match stages each of which abstracts a logical unit of packet processing. Each logical match stage allows size the match table to be configured. An input selector picks the fields to be matched upon. The result of this match is an action. More precisely, there is an action unit for each field F in the header vector, which can take up to three input arguments, including fields in the header vector and the action data results of the match. Instructions may only modify fields in

the packet header vector, update counters in stateful tables, or direct packets to ports/queues. Control flow is defined by next-table-address, which is the index of the next table to execute and it is also provided as an output by each table match. This can be used to drop a packet, implement multicast, or apply specified QoS. For example, a match on a specific field in Stage 1 could direct later processing stages to do longest prefix matching on IP (routing), while a different field could specify exact matching on Ethernet DAs (bridging). A recombination module at the end of the pipeline pushes header vector modifications back into the packet. Finally, a configurable queuing discipline can be applied to place the packet in the specified queue at the specified output port.

2.2. Quality Metrics

The objective of the simulator is to analyze and characterize various factors that contribute to network performance and examining the effect they have. The network performance can be expressed by several metrics such as delay, throughput and packet loss.

2.2.1. End-to-end Packet Delay

End-to-end delay is considered as an important parameter to analyze the performance of the network. The end-to-end packet delay is the time it takes to deliver a packet to receiving endpoint after being transmitted from the sending endpoint. The average time varies according to the amount of traffic being transmitted, scheduling policy, number of cores available at that given moment, which reflects how well does whole system scale as load increases. If traffic is greater than bandwidth available, packet delivery will be delayed.

2.2.2. Throughput

The throughput metric represent the overall work accomplished by the network. Throughput is defined as the ratio of the data packets received by the endpoint to the packets sent by the sending point. The network capacity is defined to be stable when the average number of packets inserted into the network is the same as the average number of packets reaching their destination[36]. Network throughput is important for both round-trip time and handling a large number of simultaneous transactions.

2.2.3. Packet Loss

Packet loss occurs when one or more packets of data travelling across a network fail to reach their destination. Packet loss is reflected in the throughput and delay characteristics. When a packet is dropped before it reaches its destination, all of the resources that it has consumed in transit have been wasted. A 10% packet loss, for example, reduces throughput by a barely noticeable 10% if the retransmission algorithm is implemented efficiently, but could well make an audio or video connection unusable. The first step to understanding (and fixing) packet loss is to accurately measure its existence. Loss rates are especially high during times of heavy congestion, when a large number of packets are compete for scarce sources like processing elements.

Chapter 3: NPU SystemC Model Semantics

3.1. Overview

SystemC is a widely used language for System-on-Chip design and validation. It is essentially a discrete event simulation library in C++ that provides modeling abstractions for system-level design. The transaction-level modeling (TLM) abstraction in SystemC enables creation of abstract hardware models, particularly of memories, for high speed co-simulation of hardware and embedded software. TLMs abstract away cycle-accuracy and bit-level accuracy using abstract function calls to access memories or hardware services.

NPUs are widely used in edge routers for performing high-touch functions like encryption and compression. They are also good for CRC and checksum calculation, metering, accounting and keeping statistics: operations that are best done in software. Therefore, NPU architects use a large number of multi-threaded RISC cores to exploit data parallelism, while providing the flexibility of software programming. Figure 2 illustrates the high-level architecture of a simplified NPU, inspired by the SNP 4000 architecture [34].

3.1. Basic Workflow

The NPU consists of configurable hardware units for parsing, scheduling, reordering, traffic management and deparsing of packets. It consists of multiple identical processing clusters, each consisting of a set of RISC cores, memories and hardware accelerators.

The control plane populates the match table entries in the NPU's memory via an agent. Packets arriving at the ingress are sent to the parser for classification and generation of a packet descriptor. Parser also ensures that sufficient resources are available to continue with packet processing. If resources are not available, parser drops the packet, otherwise, the packet descriptor is forwarded to scheduler for further processing.

The scheduler then assigns the packet to the processing clusters with the highest available processing credit; that is, the processing cluster currently doing the least amount of work. In the event that more that several processing clusters modules have equivalent credit, the first found receives the assignment. If no processing cluster modules are available, the scheduler will buffer incoming packets until a module becomes available. Another option includes using a push-back method to prevent incoming packets from being accepted until the high-traffic situation has been resolved. The packet processing application runs as concurrent identical threads on the CPU cores in the clusters, in a run-to-complete fashion. The application thread transforms the packet header, inspects the associated payload if needed, and then waits for the scheduler to send the next packet. Since any two packets may be processed on different cores, they may encounter different delays and may leave the processing clusters in a different order than the one they arrived. Therefore, a reorder module is used to restore the ordering of packets. The packet then goes to a traffic manager that sends the packets out according to their priority. The deparser

generates the deparsed header, and recombines the header with its payload before sending the packet out on egress.

3.2. NPU Modules

We chose SystemC as the modeling language since it provides the necessary constructs, namely concurrency, event-based synchronization, timing and object-orientation, needed to create an executable system level model. For example, a SystemC module (SC_MOUDULE) can model a hardware component, which can have its internal logic and external ports to connect to other modules. A SC_MODULE can also create the representation of threads (SC_THREAD). The C/C++ code targeted for the NPU cores is wrapped inside a SystemC thread (SC_THREAD), and instantiated inside the SC_MODULE corresponding to the core.

All modules that are part of the NPU hardware are inherit from a base class. This base class provides common functionality including:

- The ability to notify attached observers of events
- The ability to add, increment, decrement, and remove hardware counters
- The ability to read and write packet descriptors, packet data, and other types of data
- The ability for modules to be addressed by name

All modules are contained within the top-level NPU module - this module represents the NPU hardware and also provides an interface to which an external observer can connect. NPU memories are represented as passive transaction-level SC_MODULE and provide read and write service through implemented interface. Modules act primarily on two data types: packets and packet descriptors. Packets contain an ID and data and represent the data passed into and out of the NPU hardware. Packet descriptors are internally used and generated; the descriptors contain

an ID corresponding to the described packet along with a several other pieces of information such as context and class of service. Additional fields can be added to the descriptor by modules within the NPU.

3.3. On-Chip Network

Network processors provide flexible support for communications workloads at high performance levels. Designing a network processor can involve the design and optimization of many component devices and subsystems, including: (multi)processors, memory systems, hardware assists, interconnects and I/O systems. Often, there are too many options to consider via detailed system simulation or prototyping alone.

All existing NPs are multiprocessors in one way or another, having multiple PEs on a single chip. All these PEs have to operate on data that is delivered through the input data stream, and since they have to operate in parallel, they must avoid blocking each other when accessing e.g. memory buffers. Therefore, the issue of how to arrange for on-chip communication is important.

Communication architectures are characterized by their performance as well as their scalability. A crossbar provides direct access between any two nodes, but scales by the square of the number of nodes; it also occupies plenty of space, expensive silicon real estate.

Traditional interconnect architectures such as shared bus and crossbars will have difficulties scaling to today data rates while maintaining reasonable costs. Dally and Towles proposed replacing dedicated, design specific wires with general purpose, (packet-switched) network [37], hence marking the beginning of network-on-chip (NoC) era.

Interconnects are one of the key architecture in any network processor unit. The network-on-chip will provide a basic mechanism for network agents to send messages to other network

agents. For the NoC architecture, the chip is divided into a set of interconnected blocks (or nodes). A router is embedded within each node with the objective of connecting it to its neighboring nodes. The router has as many as ports needed to connect with other routers and a local port to connect with its local.

3.3.1. Routing Algorithms

Router basically consist of routing algorithm and switching techniques, which can be used in order to avoid faulty or congesting ports [38]. In our implementation, OCN is modeled as a static XY routing protocol. The detailed routing algorithm is summarized in Listing 2.

```
1: Read (destination)
2: If(destination == local_core
3:    send the packet to the local core;
4: else
5:    send the packet to the neighboring router on the y-axis (or x-
      axis) towards the destination
```

Listing 2. XY routing protocol

3.3.2. Router Modeling

In our SystemC implementation of On-chip network routers are modeled as SC_MODULE, provide transaction service between any two module instances (including memories) and routing logic is implemented as SC_THRAD. Each node has vector of interfaces, which there is an input/output port per interface. Each interface is connected to a module or another router by a pair of queues (sc_fifo channels) representing the links between routers. Routing table is the input to constructor of router SC_MODULE. The basic mechanism inside each node is illustrated in Figure 3. Predefined timing for each read, lookup and write can be specified to add timing to the OCN.

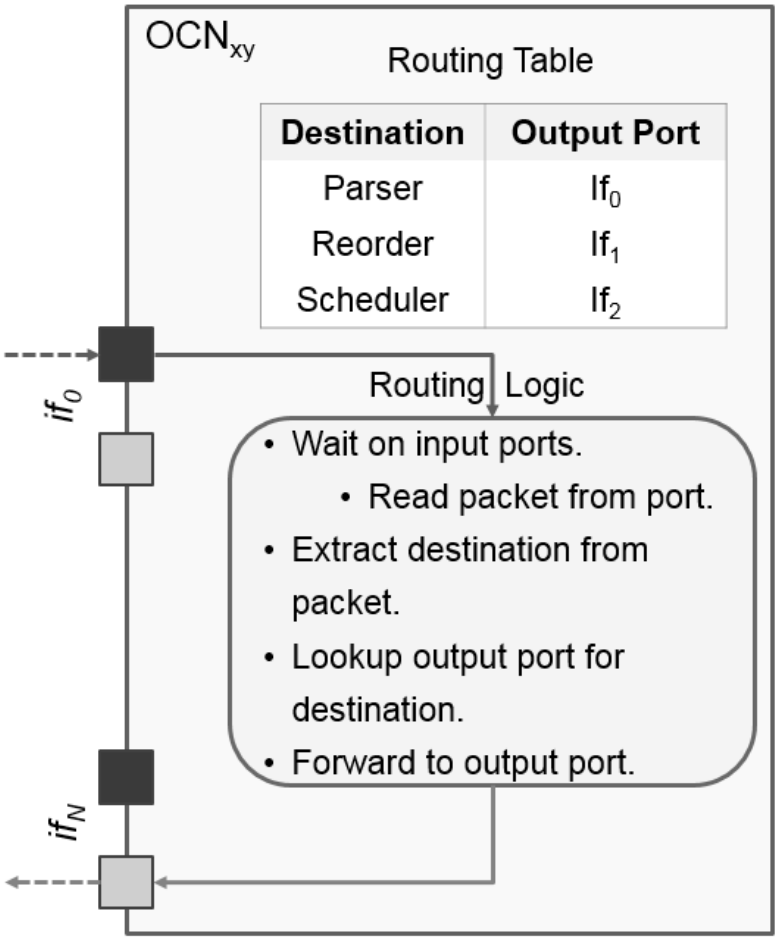

Figure 3. Router modeling in SystemC

3.4. Observers

One way to get notified of the events happening inside the model is to have the receiver part repeatedly check the sender for updates but this approach has two main problems. First, it takes up a lot of CPU time to check the new status and second, depending on the interval we are checking for change we might not get the updates "immediately".

Observers are used to solve this problem. Users may use any number of observers to examine the internal behavior of the model during the simulation. Observers must inherit from the observer interface and must implement all interface methods; however, an empty

implementation is considered valid in order to permit the user to develop small, focused observers for specific purposes.

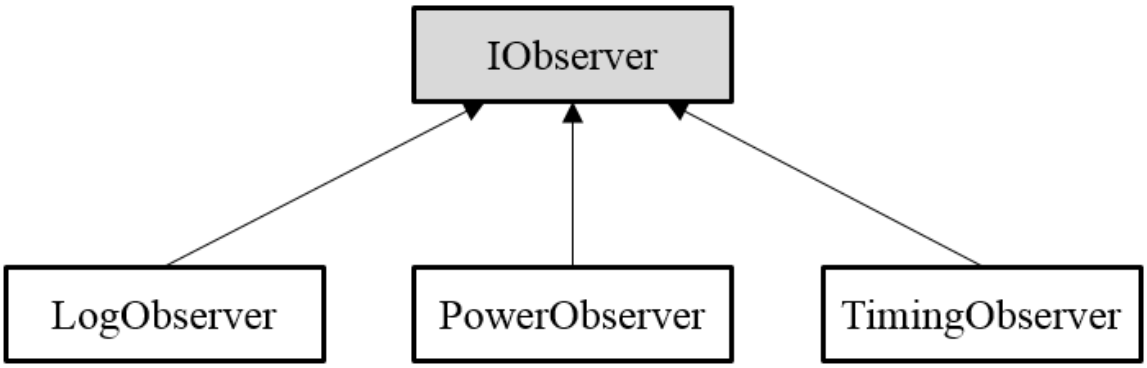

Figure 4. Example observers developed

Three observers in Figure 4 are included with the source code to serve as examples. Of these, the TimingObserver is the simplest and most illustrative of overall NPU behavior; this observer simply echoes all notifications to the standard output in a readable format. The LogObserver saves several different types of information to a file for post-processing. The ThreadObserver examines the behavior of threads within an execution unit. Users are encouraged to examine the source code for these simple observers prior to building their own.

The NPU class In addition to simulating the hardware, provides access to Observers that can connect to the NPU in order to receive data about Packet operations and thread activities. An arbitrary number of Observers can be connected to the NPU. In order to develop a new Observer, it is necessary to inherit from the IObserver interface. All functions in this interface are pure virtual – therefore, it is necessary that every Observer implement all functions. However, functions that are not required by a particular Observer can be implemented as empty functions. Sample codes for IObserver and ThreadObserver have been included in Appendix 1.

Given the number of concurrent threads and active processing elements in the model, the observer system was modified to be thread safe by introducing an event queue as illustrated in Figure 5. Observers are no longer directly notified when an observable event occurs. Instead, the event data is bound to the event function to create an event holding all the event data. This event is then pushed to the event queue. A dedicated thread in the top-level module continuously monitors the event queue; if there is at least one event in the queue, the event monitoring thread will pop the event and notify all the observers. Meanwhile, the simulation can add events to the queue without limit and without fear of invalidity or corruption.

The queue itself is a custom data structure that we previously designed for use in SystemC modeling of NPU. It is a thread-safe multi-producer/multi-consumer queue, so multiple simulation modules may add events to the queue while the monitoring thread removes events.

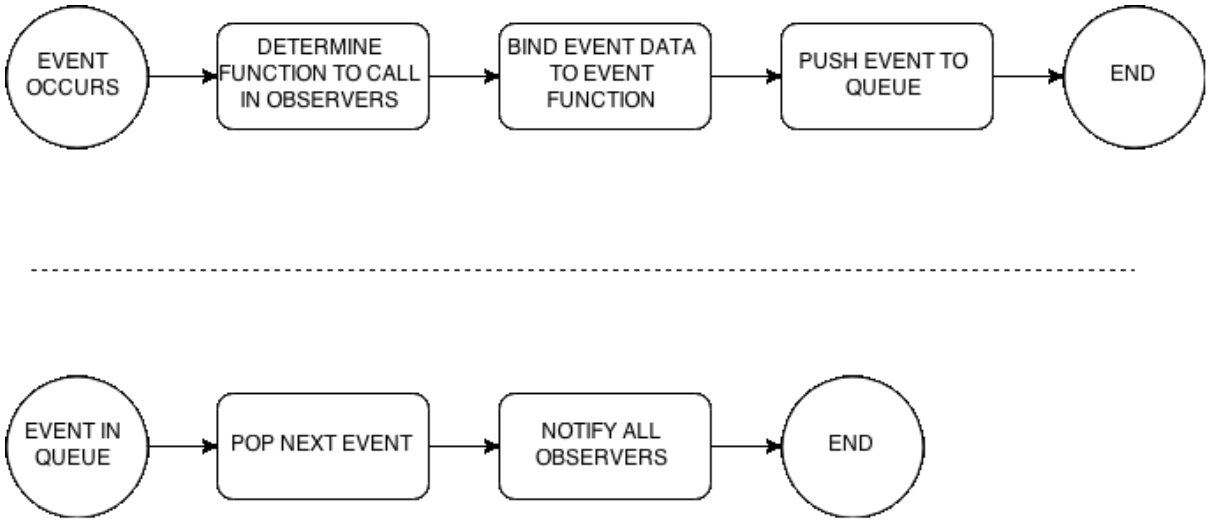

Figure 5. Thread safe observers

Observers are attached to the model at instantiation and receive notifications from the top-level module as well as all submodules. Notifications occur when data (packets or packet descriptors) are read, written, or dropped; when counters are added, removed, or change value;

when processing threads are started, paused, or completed; or when processing cores become busy or idle as can be seen in Figure 6.

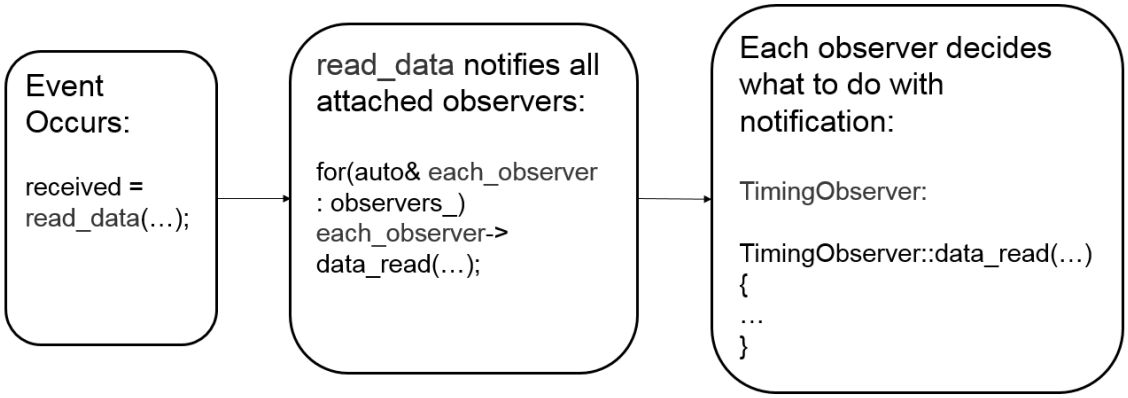

Figure 6. Observers

3.5. Functional Validation

The most obvious purpose of simulation is to check that the system performs the logical function for which it is intended. We build upon the publicly available P4 compiler to target P4 application to our forwarding device models. The original P4 compiler is targeted for a soft-switch and written entirely in Python. The compiler front end is a Python module which is included by the back end. The high-level intermediate representation is a collection of data structures in memory. The back end code generation is done using a templating library. Finally, we do not explicitly model the memory management of the target platform or the I/O needed for debugging. These services are used from the run-time system available on the host.

The C code generated by the P4 soft-switch compiler is further compiled into a set of static libraries and headers that are imported into the SystemC model of the forwarding device. Additional templates have been added to the P4 compiler back end to expose an API suitable for use in the SystemC model. These APIs, specifically, implement initialization, parsing, table

application and deparsing. The API methods are called directly from the user logic defined in the respective SC_THREADs in the SystemC model.

We evaluated the proposed method using a sample P4 application and a sample test traffic as shown in Figure 7. We created a P4 soft switch inside Mininet and used Harpoon traffic generator[39] to create test traffic. Harpoon is an application-independent tool for generating representative packet traffic at the IP flow level. As Harpoon is not topology-aware, we captured each interface's ingress and egress separately and merged ingress and egress streams chronologically to reflect the traffic over the time. Then, used ingress stream as input to NPU model and egress stream to validate NPU model output.

We created host-compiled models for various NPU architectures based on Figure 2, where we varied the number of clusters. We also developed a soft-switch implementation in SystemC, based on the generated code from the P4 compiler.

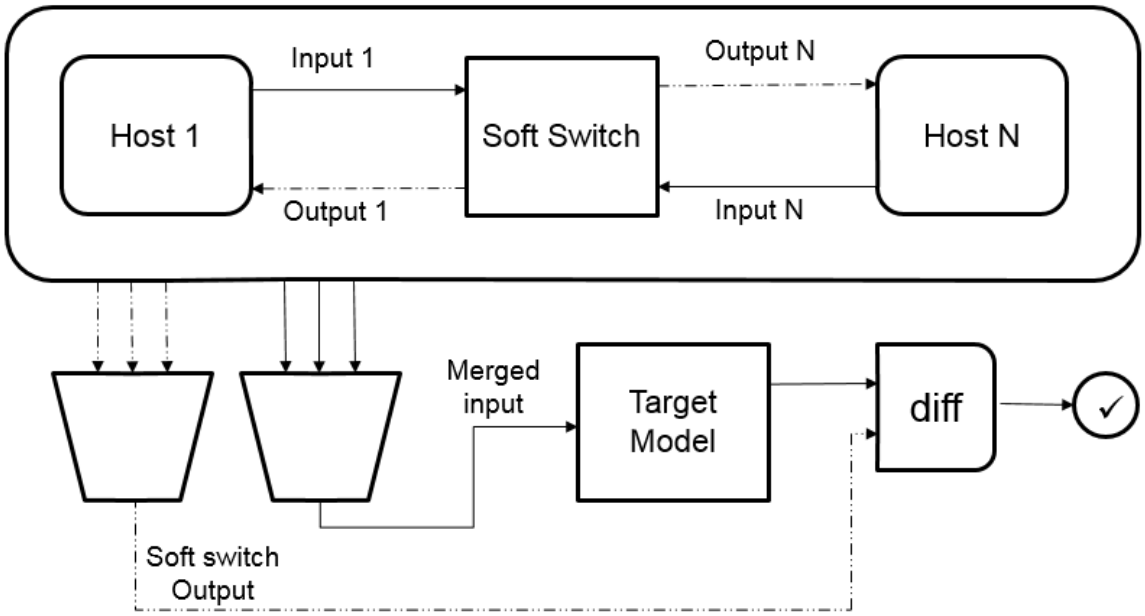

Figure 7. Model Validation

The sample P4 program is based off of the “simple_router” application provided with the P4 soft-switch compiler[40]. The program has five match-action tables and three header types (Ethernet, IPv4, and TCP). The program first performs a longest-prefix match on the destination IPv4 addresses and uses this to set egress port and next-hop address metadata fields, additionally decrementing the Ipv4 TTL field. Next an exact match is performed on the next-hop address, and the result is used to rewrite the Ethernet destination address. An exact match is next performed on the TCP source port and the result is used to rewrite the TCP source port. If the TCP source port is not matched, then an exact match is performed on the TCP destination port and the result is used to rewrite the TCP destination port. Finally an exact match is performed on the egress port metadata, and the result is used to rewrite the Ethernet source address. Additionally, the IP and TCP checksums are validated during parsing and updated during deparsing. The table size for the match tables were set to 2048.

3.6. Limitations

One limitation of this framework is that we validated our simulation functionality against reference emulation: Mininet. Emulators, like most tools, do have their drawbacks. Many of these problems can be attributed to the level of details and states that is captured during emulation which results in the computationally intensive processing required by emulators. As a consequence, the Mininet virtual network is limited by the underlying system capabilities. A higher packet forwarding capacity with support for handling higher aggregate bandwidth translates into an ability to emulate more links and routers, which will result in an unpredictable and unrepeatable behavior of the modeled system.

For example, as we increase the load of the system, the instantiated soft switch inside Mininet starts to drop packets not because of the application running on soft switch but due to host limitations like OS tasks running on the host system and user processes. The major drawback of this behavior is that performance and scalability of our reference model under heavy loads very much depends on the underlying system, which results in uncertain behaviors. Of course, powerful hardware and high capacity memories are a quick solution, but they are normally too expensive. As a designer we need simulators to be reliable enough to be trusted so architectural decisions can be made based on their output. Thus, it is important to study different emulation environments in order to find suitable evaluation frameworks to be used as the reference for functional validation of our model.

Chapter 4: Timing Estimation with Dynamic Instruction Count

The modeling framework must be equipped with techniques for ranking of different user simulation methods based on metrics such as: end-to-end delay, core idle, busy time. Without timing estimation, no end-to-end delay comparison between different architectures can be done.

Figure 8 shows that when metrics are available for comparing one design to another, Design Space Exploration (DSE) techniques can be used to perform optimization, eliminating inferior designs and collecting a set of final candidates that are further studied.

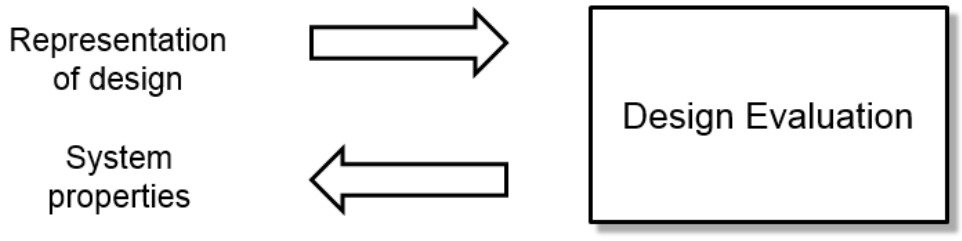

Figure 8. Design evaluation based on timing information

4.1. Host Compiled Simulation

The complexities of the multi-processor/multi-core systems-on-chip design space have made traditional cycle-accurate simulators inefficient. Cycle-based simulators are highly accurate but very slow, especially in a multicore or multiprocessor context, while at the other end of the spectrum, host compiled simulations provides fast evaluation.

SystemC supports incorporating delays into the model by using wait statements. At high level of abstraction (loosely timed), a relatively large piece of code can be executed instantaneously (e.g. processing an image or a macro-block). To model time that should have elapsed during this action, one usually use a single $wait(\Delta t)$ statement following it, where Δt is the time it would take to perform this behavior on the real circuit. By pushing computation modeling to higher abstractions, simulators can provide significant speedups, but often at the expense of limited or no timing accuracy.

4.2. Dynamic Instruction Count Pass

In order to consider the timing properties of software execution in the system-level model, the modified front-end of the LLVM compiler is used to statically determine execution time of the applications at the level of basic blocks.

A basic-block is, by definition, a sequence of instructions such that if the first instruction is executed, it is never interrupted with external instructions and all the instructions in the basic block are executed. The basic-blocks are not visible at the source level of the applications and can only be observed at the assembly level. The delay of a basic-block for a given execution is equal to the sum of the delays of all of its instructions for that execution. Resulting annotated

code is compiled and executed on the host computer to estimate performance of the complete system.

Applications are written in C or C++ and passed to the model via a callable object, such as a function pointer, containing the entry point of the application. The model runs applications to completion using a single call to the entry point. An instance of the application is executed in a single thread on a processing element; given the number of concurrent threads and active processing elements in the model, the several instances of the application may run concurrently, each operating on a different Packet.

In order to add application timing information to the model, it is assumed that the number of assembly-level instructions executed during a run of the application – that is, only the instructions that are actually called during a particular path through the application – is representative of the time required to execute the application. By adding a wait equal to the number of instructions executed during a particular run, the model is able to include valid application timing information.

The procedure as illustrated in Figure 9 is fairly simple and depends on a custom tool, called the Dynamic Instruction Count Pass (DICP), implemented as a LLVM Function Pass [41] which operates on a counter attached to each processing element:

1. The DICP finds the end of each BasicBlock, in each potential branch, of the application
2. The DICP adds an LLVM instruction to increment the counter by the number of instructions in the BasicBlock.
3. The transformed bytecode for the application is linked with the rest of the model
4. At runtime, the counter keeps the accumulated delay for each process at any given time during the TLM simulation. Each instance of the application is followed by a wait with

time units equal to the value of the accumulated delays. We do not apply sc_wait after each basic block execution because it is an expensive function that forces the SystemC simulation kernel to reschedule simulation events.

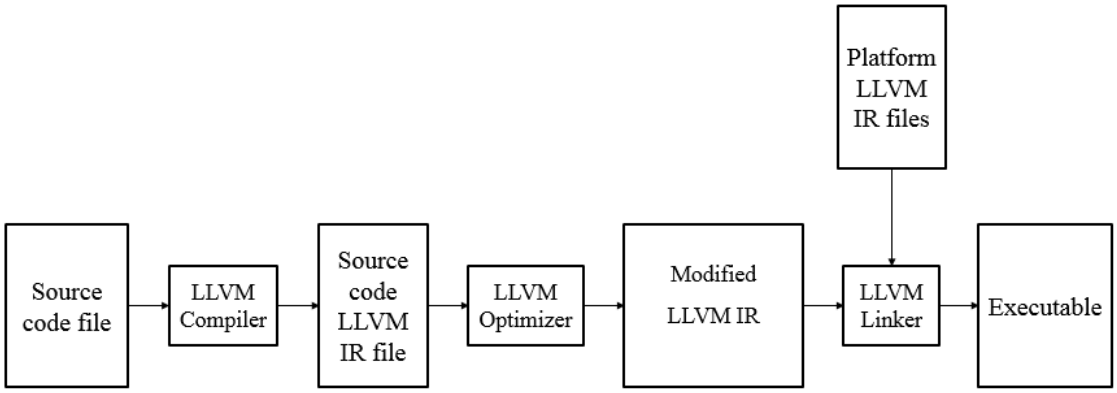

Figure 9. Dynamic Instruction Count Pass

Figure 10 shows an example of the application intermediate code before and after the procedure, which adds instruction to increment the attached counter by the executed instructions count (IC) in each basic block.

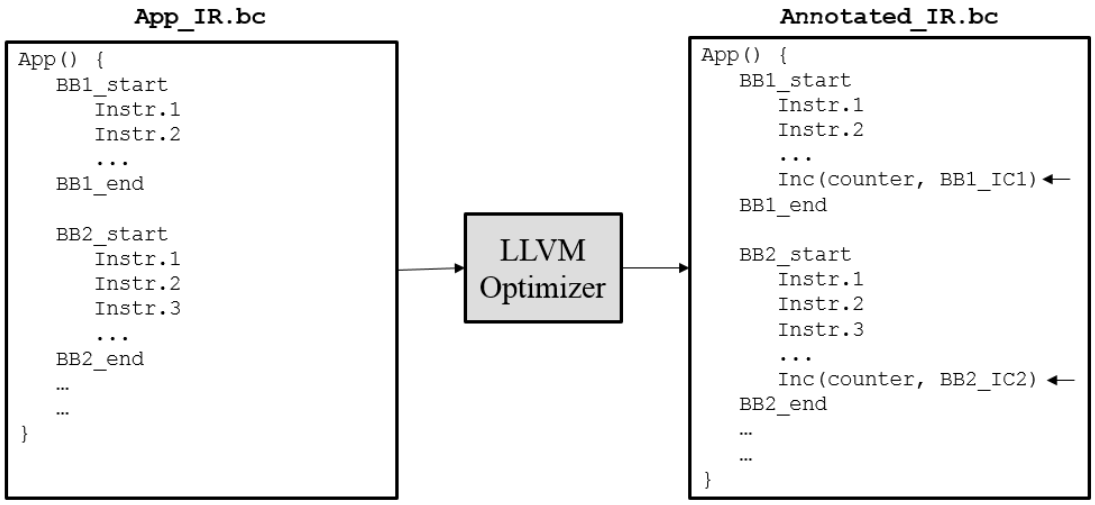

Figure 10. Application intermediate code before and after applying DICP

Using LLVM intermediate representation to calculate number of instructions rather than machine code makes this approach easy to use and fast. But on the other hand, when using LLVM IR as a compiler target, certain decisions are left to be made later by the LLVM to machine code compiler. In particular, we don't have to do exact instruction selection, instruction scheduling, register allocation (LLVM IR has infinite "registers") and memory allocation when compiling, which leads to inaccuracy.

4.3. Simulation Results

In this section, we present the simulation results of the proposed model. The objective of this section is to analyze and characterize various factors that contribute to network performance and examining the effect they have. We targeted 3 different benchmarks: end-to-end packet delay, packet loss and power consumption to measure the impact of having different architectures for network processors. For each benchmark we explore the impact of having different number of processing element versus different input rates. Therefore, NPU designers can benefit from executable models that enable them to evaluate the trade-off between the number of cores and the power consumption in their design. Such an executable model can also help with fast and early functional validation of the NPU architecture before hardware availability.

4.3.1. Average end-to-end Packet Delay

Figure 11 shows how well does whole system scale as load increases. Increasing input rate means more packets passed to ingress that network processor needs to schedule, process and move them to output. Because of limited cores that we have, at some point we have to buffer incoming packets as we cannot assign any more packets to cores. In this situation end-to-end delay for incoming packets increases.

By having more cores available inside every processing cluster, scheduler can assign packets to cores faster and prevents packets from getting aggregated, which will reduce average end-to-end delay.

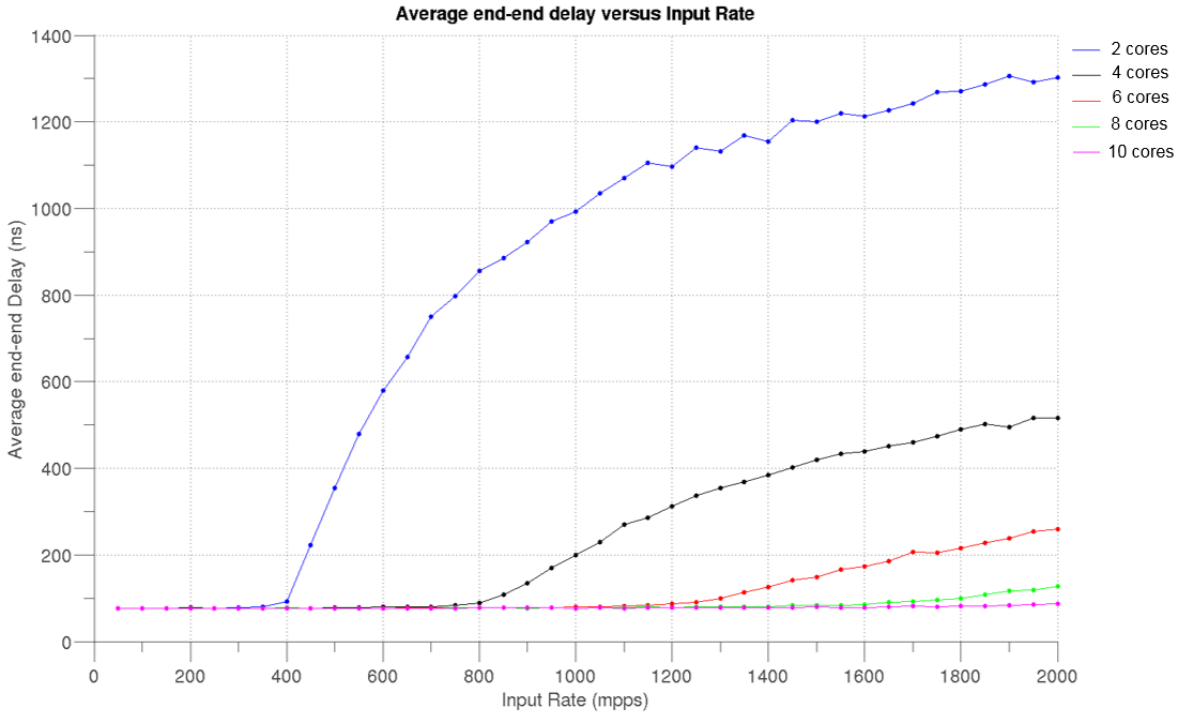

Figure 11. Average end-end delay

4.3.2. Traffic Management Policies

Packet scheduling can be used to provide (to a flow of packets) so-called quality of service (QoS) guarantees on measures such as delay and bandwidth [42]. Because of the bursty nature of voice/video/data traffic, sometimes the amount of traffic exceeds the speed of a link. At this point, what will the router do? Will it buffer traffic in a single queue and let the first packet in be the first packet out? Or, will it put packets into different queues and service certain queues more often? Or we do not buffer packets and we start dropping packets, but which packets? Suppose that we have a premium service, Isn't it better if we are forced to drop packets, we drop the

packets from less important flows? Congestion-management tools address these questions by enforcing compliance of traffic to a given traffic profile.

Traffic manager can, among other things, decide in which order packets are sent (e.g. to give priority to certain flows), decide if packets are queued or if they are dropped (e.g. if the queue has reached some length limit, or if the traffic exceeds some rate limit), it can it can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc.

In this section we evaluated effects of 3 well-known policies: FIFO, Fixed Priority and Weighted Round Robin.

1. FIFO: In its simplest form, FIFO queuing involves storing packets when the network is congested and forwarding them in order of arrival when the network is no longer congested. FIFO is the default queuing algorithm in some instances, thus requiring no configuration, but it has several shortcomings. Most importantly, FIFO queuing makes no decision about packet priority; the order of arrival determines bandwidth, promptness, and buffer allocation. FIFO queuing was a necessary first step in controlling network traffic, but today's intelligent networks need more sophisticated algorithms.
2. Fixed Priority: As the simplest example of a multiple queue-scheduling discipline, consider fixed priority. For example, imagine two outbound queues, one for premium service and one for other packets. Imagine that packets are demultiplexed to these two queues based on a bit in the IP Type Of Service (TOS) field. In fixed priority, we will always service a queue with higher priority before one with lower priority as long as there is a packet in the higher-priority queue. This may be an appropriate way to implement the premium services. It was designed to give strict priority to important traffic.

3. Weighted Round Robin (WRR): During each round from each flow that has data to send, send a number of packets equal to the flow's weight. WRR ensures that queues do not starve for bandwidth and that traffic gets predictable service.

To observe the effect of having different policies, we created a random input log of 100 packets, and used this log for three scheduling policies. We assumed having 8 different priorities in our model, with 0 as the highest priority. Each blue dot in Figure 12 represents when a packet is received at ingress (nanosecond).

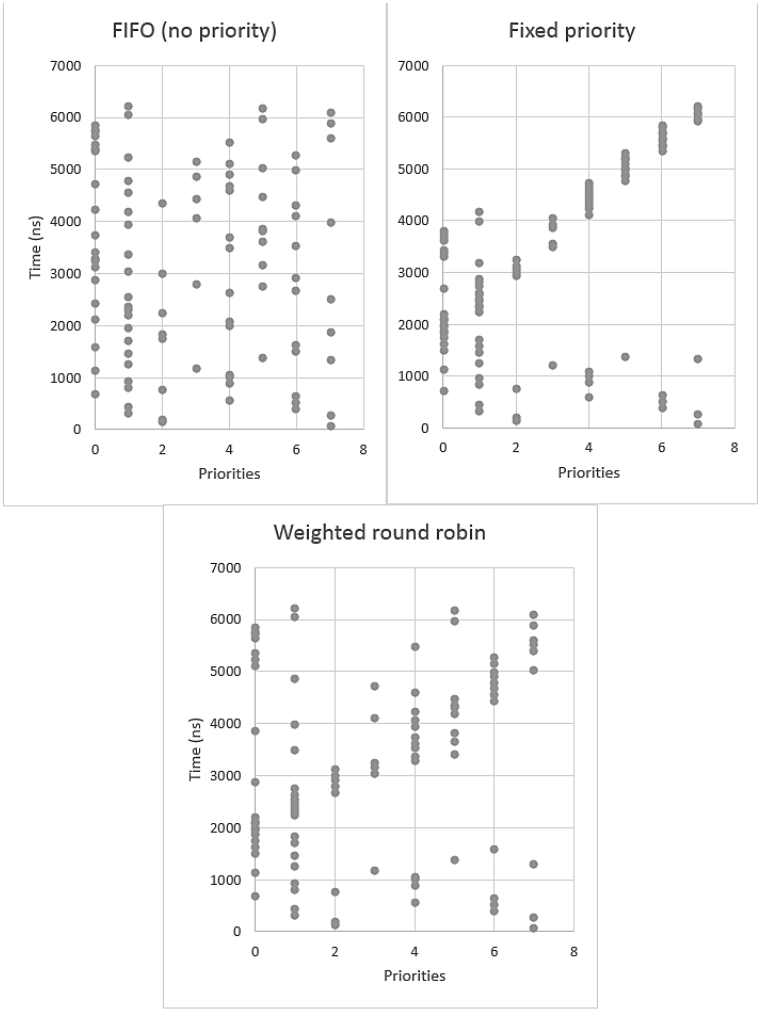

Figure 12. Scheduling policies

When we choose FIFO the packets reach to egress in the same order they are received by traffic manager, since there is no traffic prioritizing. But, fixed priority policy, ensures that important traffic gets the fastest handling at each point where it is used. So, traffic manager keeps delaying packets with lower priorities as long as there are packets in queues with higher priority. Weighted round robin on the other hand is more fair and in each round tries to have a diversity of all queues based on their weights.

4.3.3. Packet Loss

Packet loss occurs when one or more packets of data travelling across a network fail to reach their destination. Packet loss is reflected in the throughput and delay characteristics. When a packet is dropped before it reaches its destination, all of the resources that it has consumed in transit have been wasted. A 10% packet loss, for example, reduces throughput by a barely noticeable 10% if the retransmission algorithm is implemented efficiently, but could well make an audio or video connection unusable.

In Figure 13, we show the relationship between the number of processing elements and drops for different input rates. Loss rates are especially high during times of heavy congestion, when a large number of packets are compete for scarce sources like processing elements. As described in model architecture, drop decision is made on the input congestion from the parser. When the resource usage exceeds the parser usage threshold configuration, the packet will be dropped. Parser would make accept/drop decision per each class of service based on resource usage of each packet. We have a defined resource available inside every isolation groups for every class of service. As number of processing elements increased packets, more packets get processed and the credits for each isolation group and class of services return faster, this way the number of drops is reduced.

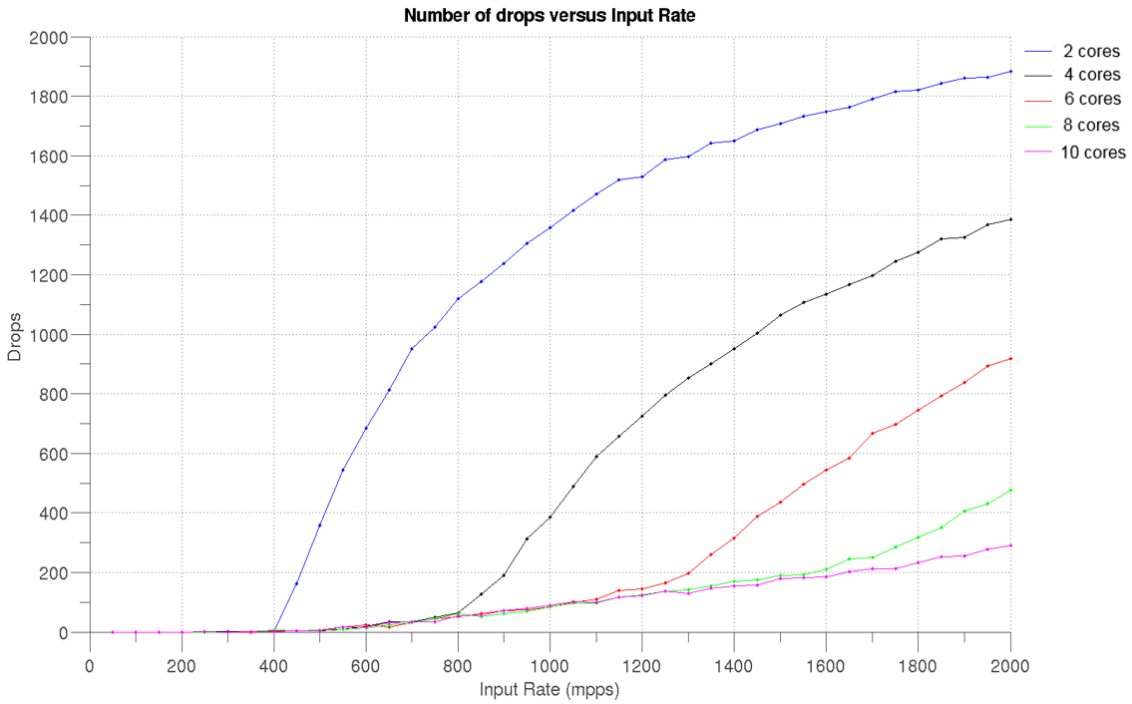

Figure 13. Number of drops versus Input rate

4.3.4. Power Consumption

Power consumption is a key issue in the design of network processor units. We observed that under low incoming traffic rates, processing elements in network processor are idle for most of the time but still consume dynamic power. If some of the processing elements are idle, it means that there is more processing power than required by the incoming packets. We designed techniques for detecting idleness of threads to reduce power consumption. We observed the idle time of a processing element during which the processing element does no useful work and waits for incoming packets.

To investigate the effectiveness of the number of processing elements on power consumption, we measured the total idle time of the cores per design for different input rates according to equation 1, and the results were plotted in Figure 14. As we expected, this plot shows that designs with more cores have more idle time.

$$N\ cores_{idle_{time}} = core1_{idle_{time}} + core2_{idle_{time}} + \cdots + coreN_{idle_{time}} \quad (1)$$

For example the idle time for the design with 4 cores is computed by the equation 2:

$$4\ cores_{idle_{time}} = core1_{idle_{time}} + core2_{idle_{time}} + core3_{idle_{time}} + core4_{idle_{time}} \quad (2)$$

We can conclude from this plot that when input rate is low, it is not necessary to keep all the cores fully powered all the time. Software configuration can be added to put out some cores into a low-power "sleep" mode whenever possible while maintaining low packet end-to-end delay.

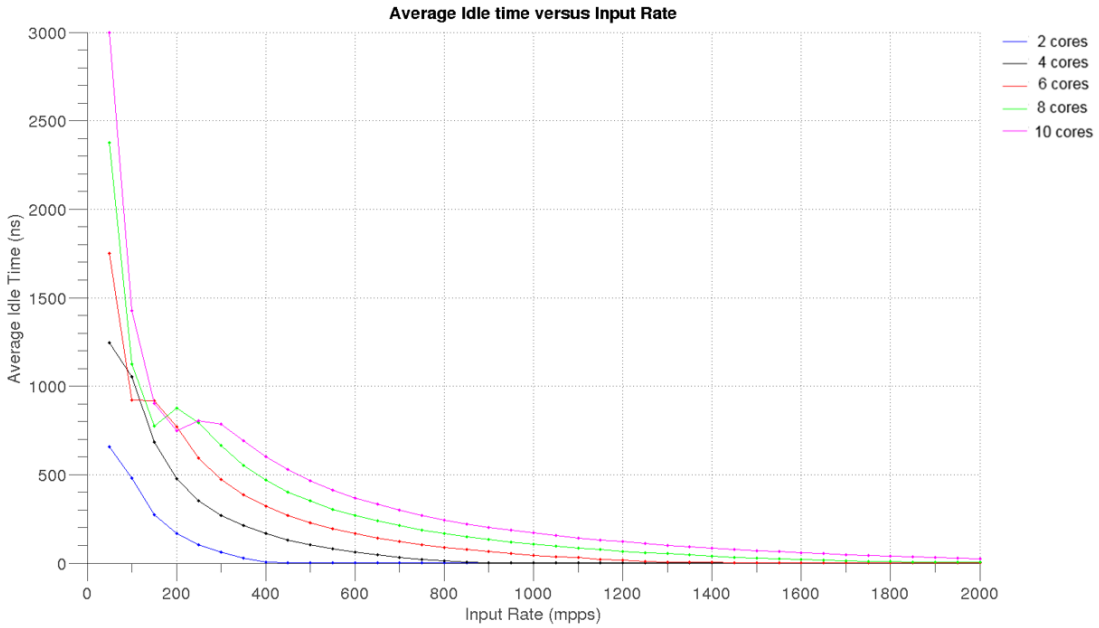

Figure 14. Average idle time versus Input rate

4.3.5. Simulation Speed

One of the major concerns when using simulators is simulation time. Simulation time is the time it takes to run a full simulation cycle, including buffering, scheduling the packet and processing. RTL models, which contain micro-architectural details are cycle-accurate but they are too slow for large scale systems and too low-level (i.e. they require a detailed implementation) for effective design space exploration. Furthermore, constructing such optimized simulators is a

significant task because the particularities of the hardware must be accounted for [7]. For this reason, it requires too much effort to build them, and these simulators are inflexible. To overcome these shortcomings, the use of higher level models for design space exploration and early software development is critical. A better solution for this problem is to use instruction-set simulator (ISS) as explained in more details in section 5.1.

An "Instruction Accurate" (IA) model eliminates much of the data processing and abstracts the timing to instruction execution ordering. We compared the simulation speed of our proposed method with that of an instruction set simulator: OpenRISC[43].

OpenRISC is a project to develop a series of open source instruction set architectures based on established reduced instruction set computing (RISC) principles. It is the original flagship project of the OpenCores[44] community. The first (and currently only) architectural description is for the OpenRISC 1000, describing a family of 32 and 64-bit processors with optional floating point and vector processing support.

In order to evaluate the speed of the proposed method, different applications with different number of basic blocks are used. Then, execution speed of application in our annotated approach is compared to that of the instruction set simulator and speedup is calculated. Experimental results in Table 1 shows an average of 40X speedup compared to OpenRISC.

Table 1. LLVM back-annotation tool speedup

	Number of BasicBlocks	OpenRISC	DICP	Speedup
App1	1	1806176	39548	44.6704764
App2	5	566074	17023	32.2534806
App3	10	829842	25668	31.329827
App4	20	1133234	31231	35.2855496
App4	50	2134221	45645	45.7569504
App5	100	6453495	128342	49.2835783

Chapter 5: Timing Estimation with Instruction Set Simulator

5.1. Overview

Technological progress causes complexity of systems-on-a-chip (SoCs) to grow rapidly in last few years. Millions of logic gates and sophisticated analog circuits, very often hundreds of thousands lines of source code, that is intended to work under control of operating system [45]. The challenge of designing and testing software earlier in the design process is becoming an increasingly significant factor, especially in cases where the hardware environment may be extremely complex or not yet available. Approaches can generally be classified according to the level of granularity at which target functionality and timing is modeled. There are two standard ways:

- Using host compilation
- Using RTL models

These methods have drawbacks – in host compilation as demonstrated in last chapter since estimation is performed before runtime, simulation speed does not suffer but it does not give any architecture-dependent details, while RTL models are slow, often multiple orders of magnitude slower than native execution. Consequently, for fast system-level performance analysis, the abstraction level of processor models must be raised. The better solution for these problems is to use instruction-set simulator (ISS). The proposed method relies on the integration between an instruction set simulators (ISSs), Open Virtual Platform and the SystemC simulation environment which contains other components (Ingress, Parser, etc.). The aim of this work is to provide designers with the possibility of faster and efficient architecture exploration at a higher level of abstractions, starting from an algorithmic description to implementation details.

5.2. Instruction Set Simulator

Instruction set simulation is a software technique that mimics the behavior of executing binary instructions on the target processor. The processor on which the binary instructions should run is called the target processor, while the processor on which the ISS runs is called the host processor.

An instruction-set simulator is a tool that runs on a host machine by simulating the effects of each instruction on a target machine, one instruction at a time. Instruction-set simulators are indispensable tools in the development of new programmable architectures. They are used to validate an architecture design, a compiler design, as well as to evaluate architectural design decisions during design space exploration. Instruction set simulators are attractive for their

flexibility: they can in principle, model any computer, gather any statistics, and run any program that the target architecture would run.

5.3. SystemC/OVP Simulation

ISS functionally behaves as model CPU, but it does not map any internal architecture and thus – it simulates four to five rows faster, compared to the RTL. Each ISS contains cache memory, private local memory, and the minimum set of units required to perform basic functionality. They can provide detailed functional information, such as register values, the program execution time.

Since ISS is intended to control rest of the SoC, it must be equipped with appropriate interfaces to communicate other components. As the solution, TLM 2 standard was chosen to be implemented alongside with ISS modules. Transaction-Level Modeling (TLM) has become tremendously popular and almost universally accepted as a vehicle for acceleration of platform model integration by having different level of abstractions inside the model.

5.3.1. OpenRISC

There are some limitations that make the use of OpenRISC impractical as an instruction set simulator:

- The lack of support for C++ and multicore processors
- OpenRISC 1000 is the only architectural description available and no support for other processor architectures like: ARM, PowerPC, etc.
- The fact that OpenRISC OR1k is no longer maintained.

5.3.2. Open Virtual Platform (OVP)

Open Virtual platforms is a simulation framework marketed by Imperas [46]. The simulator provides APIs for the modeling of processors, memory, sub-system models and platform functions, without concern for detailed simulation operation. OVP includes the Instruction Set Simulator (ISS). ISS allows to run compiled binary embedded software elf files on specific embedded processor variants at hundreds of MIPS without the need to develop a virtual platform.

OVP models are used by platforms as shared objects dynamically loaded at simulation time by the OVPsim runtime. To integrate OVP with SystemC, native interfaces (wrappers) of CPU and all OVP peripherals are provided.

A. Platform construction

To use OVP models, SystemC must instantiate one “tlmPlatform” object. This object keeps the quantum period which sets how long each processor model instance waits before running again. Each processor model instance keeps a figure which controls the effective number of instructions per second (IPS) executed by the model. It uses this and the quantum period to decide how many instructions to run in each quantum. The default quantum period is 1mS. The default IPS is 100,000,000. Thus, by default, a processor runs 100,000 instructions per quantum (this matches OVPsim’s internal scheduler used in a non-SystemC environment) [46].

B. Processor and Peripheral Models

Each processor model is run from a SystemC thread. The thread executes IPQ instructions on the processor without advancing SystemC time. Each instruction may or may not cause TLM2.0 transactions to be propagated to other components in the platform. When the allotted instructions

have completed, the thread calls SystemC $wait()$ to advance time. Thus each processor executes a number of instructions at a time in a round-robin schedule. OVP peripherals can also have a wrapper layer to enable their use in SystemC TLM2 platforms.

C. Integration Implementation Details

The main drawback of SystemC/OVP model is the fact that OVP is designed to execute the application once and exit, thus to integrate it inside our packet flow, which for every incoming packet, application needs to be executed, we have to use call back functions.

Figure 15 details the architecture of the OVP/SystemC Processing Element (PE). The numbers in the Figure correspond to a packet reception, and its processing by the OVP CPU model. In the first step, the PE receives a packet from its Network Interfaces (NI). By writing the packet, an event is triggered notifying the receiver module (block inside of the SystemC-OVP interface) about the incoming packet. The receiver module then reads the incoming packet, stores it into a buffer used to synchronize the communication between the untimed CPU and the ISS (step 2). The module informs the CPU that there is data stored in the buffer. A memory mapped register is used to alert about the stored data. OVP CPU polls this register periodically. Once the CPU is ready, the data is read.

The data embedded in the packet, is read through a DMA module using memory mapped registers (register bank in step 4). The register bank is implemented using an external memory, mapped in the processor address space, where each register has a pre-defined address. The CPU is connected to a bus to which all address-mapped components are connected. This bus connects the local memory and the register bank.

Callback functions are executed on every read (regbankRead) or write (regbankWrite) access to the defined address area. Listing 3 shows an example of a callback function related to a read memory access. The callback function name (regbankRead) is defined as a parameter of the ICM_MEM_READ_FN macro.

```
ICM_MEM_READ_FN (regbankRead)
{
     …
     buffer = NI->read();
     icmWriteProcessorMemory (address, buffer, bytes);
     …
}
```

Listing 3. Read callback function

Once a read memory access is triggered, the value is read by sending, for example, a read data request to the processor module NI, and writes it into CPU memory. Using the memory mapped registers as interface, the CPU receives and processes data. Function parameter provides the memory address (address) accessed by the CPU, as well as the value (buffer) to be read from this address and number of bytes to transfer.

Listing 4 gives an example of a write callback function. This function is specified using the ICM_MEM_WRITE_FN macro, and the callback function regbankWrite as a parameter. This parameter provides the memory address (address) accessed by a CPU, as well as the value (value) to be written in this address.

```
ICM_MEM_WRITE_FN (regbankWrite)
{
     …
     icmReadProcessorMemory (address, buffer, bytes);
     buffer = NI->read(buffer);
     …
}
```

Listing 4. Write callback function

When the OVP CPU needs to send data through the processor module NI these steps are taken. First, the CPU uses the memory-mapped registers as an interface to the communication protocol with the processing element NI. For each register access, a memory callback is triggered, generating a SystemC event. Then, the send module receives the packet and stores it in the buffer. When the packet is completely stored, it is sent through the NI.

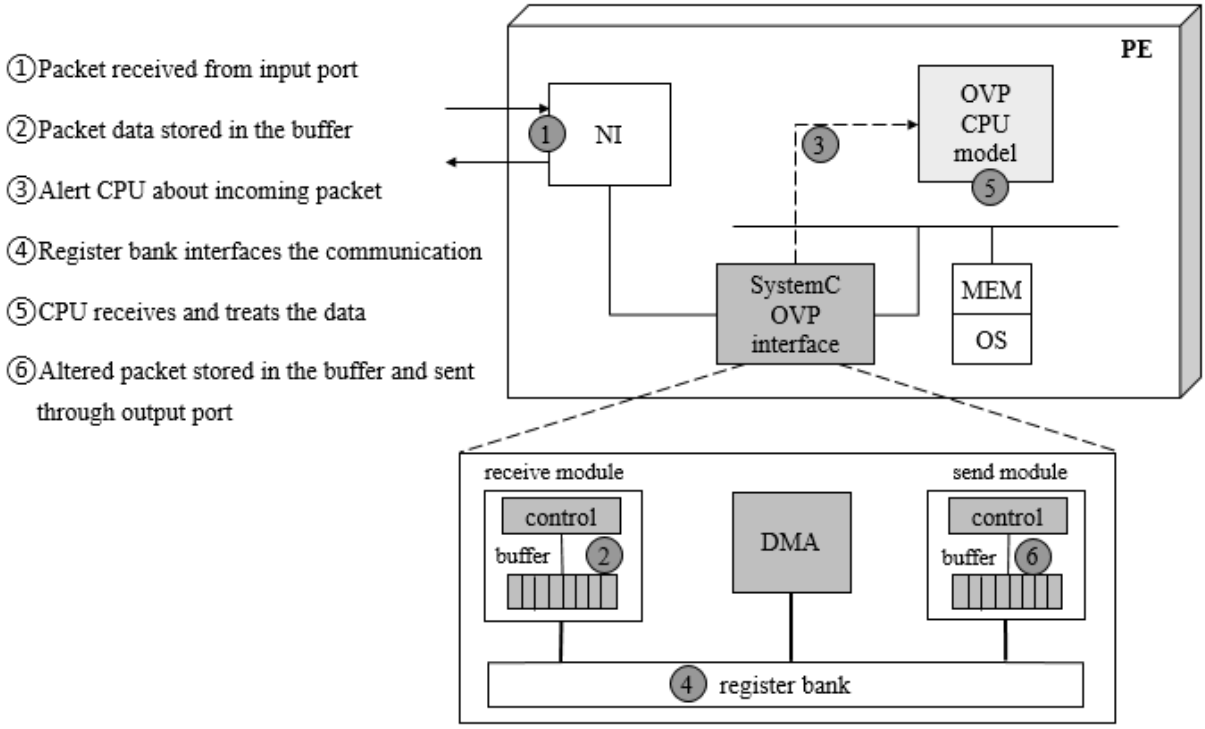

Figure 15. Integration of SystemC model with OVP CPU model

This approach makes it possible to use this model as the reference for both functional validation and adding instruction-accurate timing to our host compiled model. For more details refer to Appendix 2.

5.3.3. Simulation Results

Due to the technology advancement and our daily life being highly dependent on the Internet, The fundamental functionality to support high speed communications on the Internet are increasing in importance with the ever-growing traffic size. With increasing number of hosts and sessions in the Internet, the processing of packets at the routers has become a major concern. One of the key technologies is IP routing table lookup, which needs to be extremely fast [9].

A. IP Lookup

A fully associative memory, or content-addressable memory (CAM) [47], can be used to perform an exact match search operation in hardware in a single clock cycle. However, departure from a TCAM is an approach worth considering for the two reasons: First, TCAM has issues in power consumption and heat. Second, the advent of Network Functions Virtualization (NFV)[48] may make the use of TCAM impossible, since the virtualized network functions are currently implemented in software without TCAMs. Therefore, it is desired to implement a software high-speed IP router only with general purpose computers; i.e., personal computers (PCs), or commercial off-the-shelf (COTS) devices.

A trie structure is a general-purpose data structure for storing strings and convenient way to represent the prefixes in the forwarding table. Each string (prefix) in the routing table is represented by a leaf node in the trie. The introduction of Classless Inter Domain Routing (CIDR) has reduced the size of forwarding tables, but the lookup procedure is more complex because exact matching is replaced by longest prefix matching. The longest prefix search operation on a given destination address starts from the root node of the trie. The branching decisions are made based on the consecutive bits in the prefix. A trie is called a uni-bit trie if

only one bit is used for making branching decision at a time. The prefix set in Figure 16(a) corresponds to the uni-bit trie in Figure 16(b).

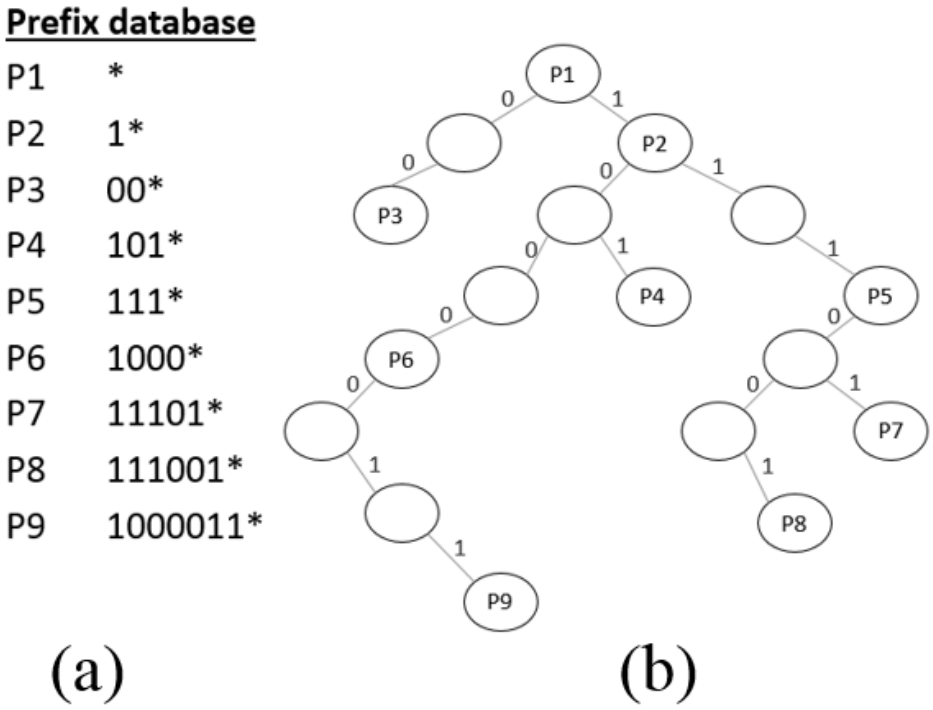

Figure 16. Longest Prefix Match

B. Testbed

We used a testbed to create rigorous, replicable testing scenarios. Client and server instances are synchronized under six stages. At every connection stage, the node defines the transaction protocol. An abstract modeling of the TCP protocol is implemented, which clients-server connection is established using three-way handshake. Client sends file request and receives the file response, which can be a file ID, a unique code or the ACK for the last transaction. Response can be the expected file size, bandwidth, etc.

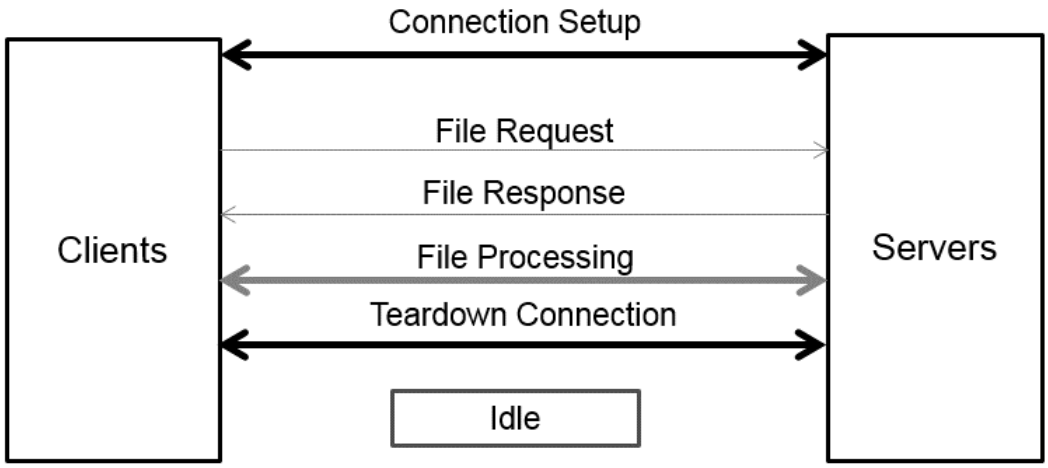

Figure 17. Connection modeling between clients and servers

Based on the architecture shown in Figure 2, the control-plane is responsible for building the IP routing table inside memory. Once configured, the control plane can start populating each table, for example by adding IPv4 forwarding entries.

Routers do two things: participate in routing protocols with their neighbors, and forward packets. We are concerned primarily with the latter. At first sight, forwarding is straightforward:

1. Router gets packet.
2. Looks at packet header for destination.
3. Looks up routing table for output interface (if an appropriate entry is not populated in the forwarding table, then it is forwarded to default node).
4. Modifies header (TTL, IP header checksum).
5. Passes packet to output interface.

C. Simulation Results

The bulk of the packet processing on an NPU takes place in the processing cluster, and the majority of the packet latency results from the search-and-lookup operations in memory. We present preliminary experimental results in this section. This experimental setup uses different data sets created by our testbed, which is described in previous section and single processor as our CPU model. We divided the evaluation in two parts. The first part consists in inducing the results from integrating our framework with OVP as instruction set simulator. We found out that although this approach achieves very good speed ratio compared to running the application code on the target machine, still not suitable to use with high intense data rates. The second part consists of applying an automatic back-annotation method, which also verifies functional validation of the annotated model. This time we measured the percentage of the speedup gained versus accuracy error. We used a 2.7 GHz Intel dual-core with 8GB RAM as our simulation host. The procedure to integrate OVP in our framework techniques is described in section 5.3.2.

The ‘Simulated instructions’ will vary depending upon the application being executed, this count indicates the number of simulated processor instructions for the processors in the platform. Figure 18 shows the relation between number of entries in forwarding tables and simulated instructions. The more entries in the table increase the possibility of matching more entries, which results in deeper traverse of the created search tree in the memory and more instructions to be executed.

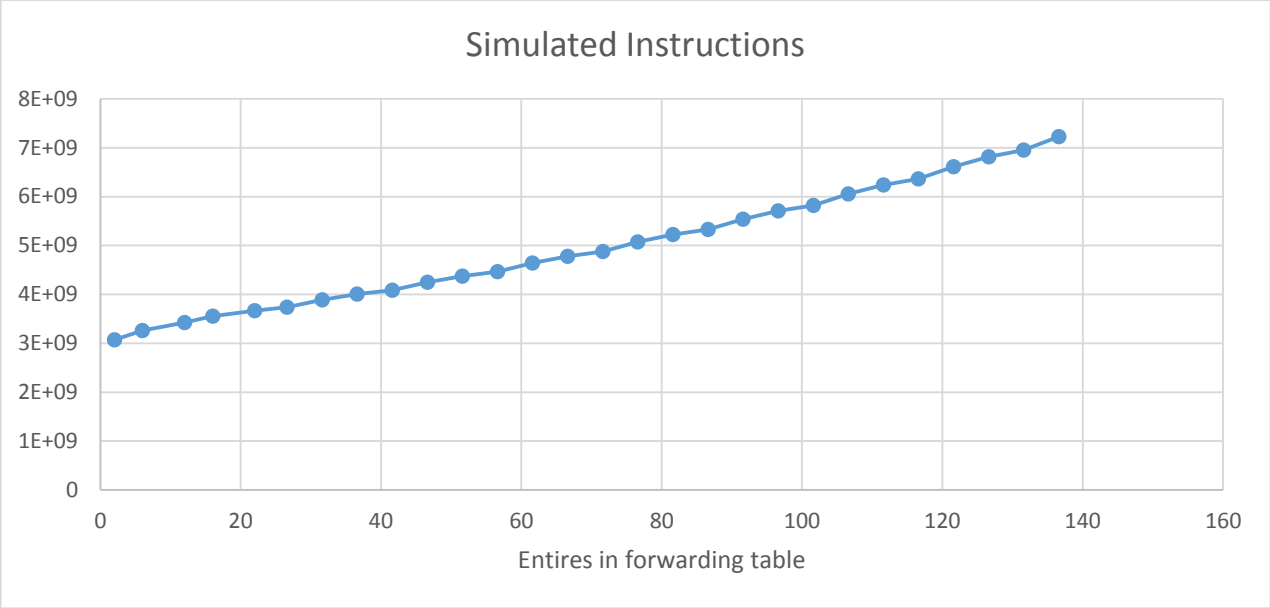

Figure 18. Simulated instruction/ forwarding table entries

As it can be seen in Table 2, each data set consists of various number of packets and 'Simulated instructions'. The real time ratio indicates how much faster this method is able to execute application on the native host machine compared to running on the real silicon based.

Table 2. SystemC/OVP Simulation evaluation

Number of Packets	**Number of Simulated Instructions**	**Real Time Ratio**
650	43766667	47.15X
1000	67247053	48.83X
8000	573851039	54.23X
26000	1878581489	54.46X
50000	3838669233	57.41X
104000	7492226409	46.53X
208000	14982744337	41.79X

By integrating this instruction set simulator, we achieved a very considerable average of 50X speedup, which will result in a very faster simulation. To illustrate this speedup further, we can consider the case of 50000 packets, which takes up to 42 minutes on target chip, compared to

under 2 minutes simulation time on the host machine. However, the time needed for trie-based lookup mechanisms grows very fast as the input load increases as shown in Figure 19, making simulation speed a premium metric when evaluating different methods.

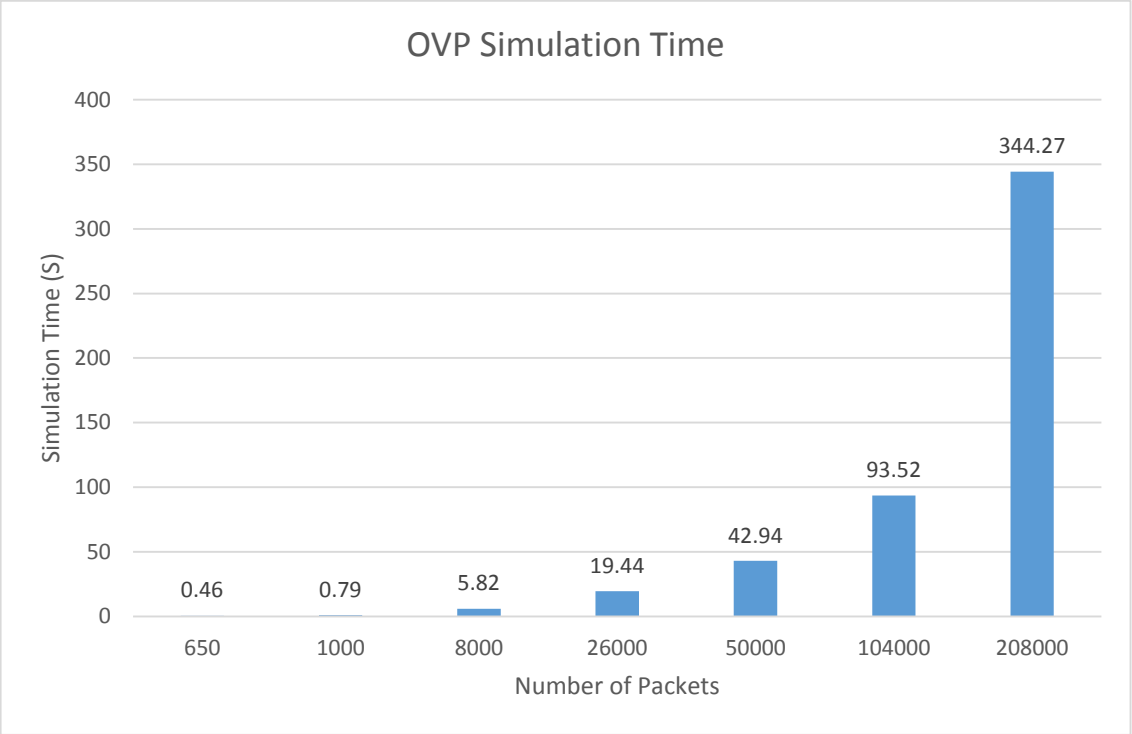

Figure 19. OVP increasing simulation time as input load increases

To understand the problem of high-speed lookups, let's study what an example internet core router has to do today. It needs to handle minimum-sized packets (e.g., 40 or 64 bytes depending on the type of link) at link speeds of 40 Gbits/s, which gives the router a window of 8 ns (for 40-byte packets) or 14 ns (or 64-byte packets) to make a decision on what to do with the packet.

In general, ISS simulators are written to test concepts and processor design tradeoffs, where flexibility is important and speed is not of primary importance. Another disadvantage of using OVP simulation is the limited available memory that can be used as CPU local memory

that makes it impossible to simulate scenarios which require large amount of memory (e.g. storing the whole forwarding table in the memory).

5.4. Back-annotation using ISS

Back annotation is used to refine timing model in a model. Back annotation is the process of including externally derived data into a design or model. This timing data is derived from several sources such as timing analyzers, delay calculators, and library specific data tables. The advantage of using back annotation over directly computing delay information in the simulation model is that the process can be decoupled from the functional design, and tools which are specifically engineered for timing analysis can be used, specifically for the fixed function elements. Although, that might be impractical to expect the software developer to provide timing data for the code that is expected to run in the processing threads. We provide a timing annotation utility that automatically inserts timing in the application source.

Host-compiled techniques, which have been introduced recently, are based on a native execution of the target code and employ pre-estimated timing annotations instead of detailed modeling of the microarchitecture. Consequently, these methods achieve higher simulation speed than ISS and hence they are very attractive for system-level simulation of multiprocessor architectures.

5.4.1. Direct Annotating

As the workflow of back-annotation is illustrated in Figure 20, one of the central idea of host-compiled simulation is to move the estimation of instruction timing from simulation time to compile time and, thus, to improve simulation efficiency.

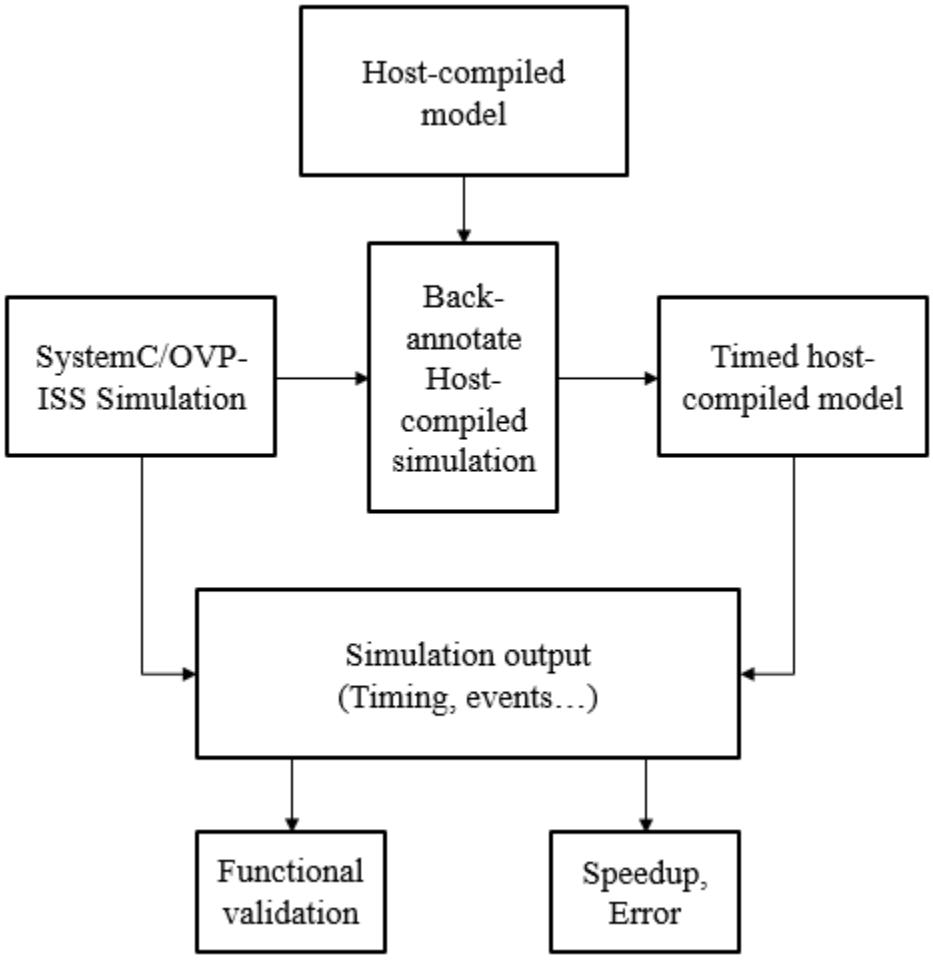

Figure 20. Workflow of experiments

The contribution of the proposed method is to provide a very fast host-compiled equivalent of the described SystemC/OVP instruction set simulator, without sacrificing the accuracy of the model.

5.4.2. Corrective Feedback Annotation

Simulating the expected line speed of the Internet core routers of 100 Gbps or even terabits per second in the near future with hundreds of thousands of entries in forwarding tables is not possible. A practical solution as can be seen in Figure 21 would be to use a subset of input to tune the parameters that most contribute to the simulation timing like number of entries in the

forwarding tables, number of input packets, length of the prefixes to match, etc. By doing so, a learning system can be trained, which can be used to back-annotate the host compiled model without the need to run the time consuming instruction set simulator. Nevertheless, such models can provide very rapid feedback to drive initial pruning of the design space of clearly infeasible solutions.

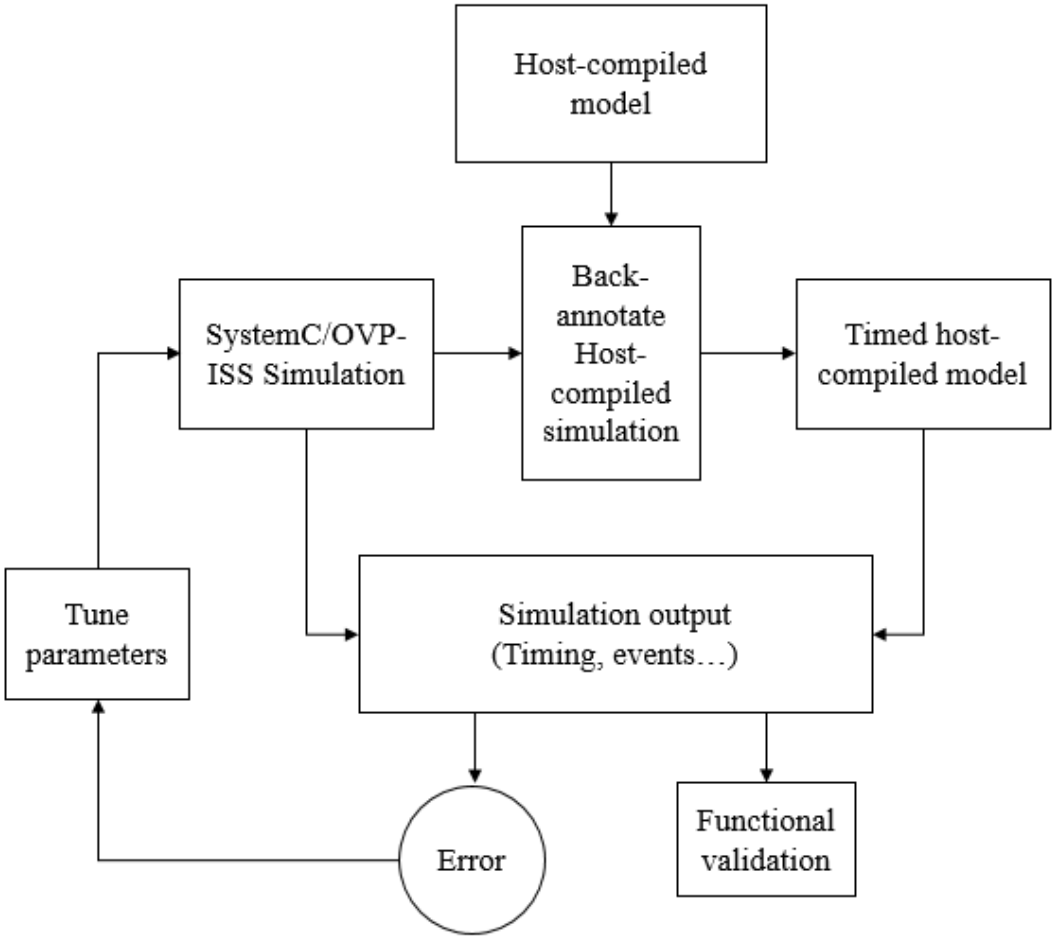

Figure 21. Corrective feedback annotation to tune parameters

5.4.3. Simulation Results

The fastest instruction-set emulators dynamically translate instructions in the target ISA to instructions in the host ISA, and optionally annotate the host code to produce address traces.

Because these emulators perform translation at run-time they gain some additional functionality, such as the ability to trace dynamically linked or dynamically compiled code. This additional flexibility comes at some cost, both in overall execution slowdown and in memory usage as can be seen in Figure 22 and Figure 23.

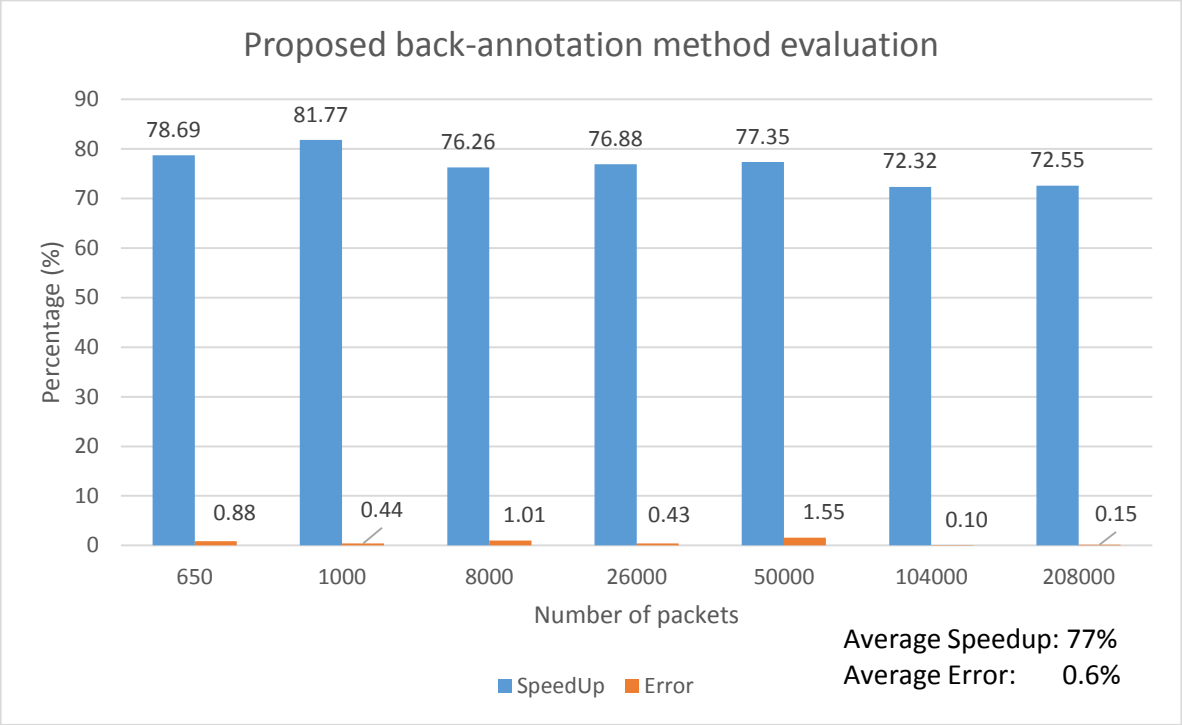

Figure 22. Proposed back-annotation method evaluation

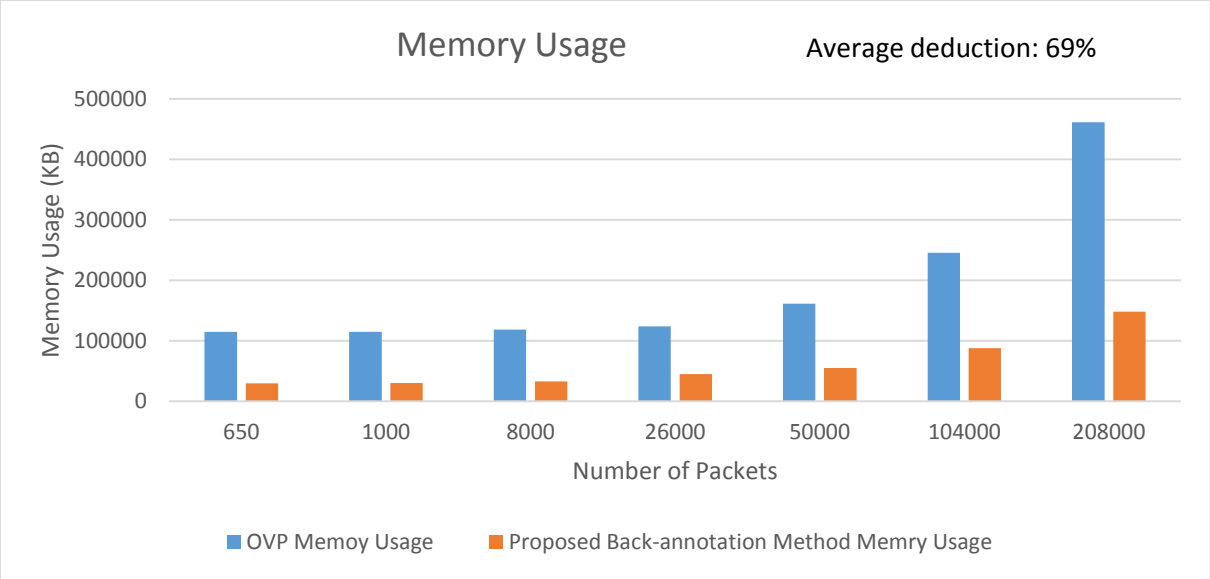

Figure 23. Proposed back-annotation method memory usage

As one could expect by increasing the number of packets, more instructions need to be executed which translates to more memory usage. Since the routing entries today are increasing super-linearly (in 2008, 256000 entries and now it has passed 512000 routes [49]), the need for a platform that uses memory optimally is an important concern.

As discussed in section 5.3.3, we used back-annotation to increase the simulation speed, while keeping the accuracy acceptable enough to make architectural decisions. The proposed method achieved an average of 77% speed up with only 0.6% error and also 69% reduction in memory usage.

This method provides assistance in identifying problems early in design process. It is accurate and fast to evaluate different design options. A design option is an element in architectural model, which could be changed to improve performance properties. In the next section we evaluate different number of cores along with different memory sizes as various design options for a network processor unit.

5.5. Timing Estimation Summary

In this section we started by host compiled simulation, which has no notation of timing, then we used a custom toolset which adds timing based on the number of instruction in intermediate representation of the source file at basic block level. Then we moved to a more accurate but slow approach by using instruction set simulators which relies on the integration between an instruction set simulators (ISSs), Open Virtual Platform (OVP), and the SystemC simulation environment. Finally we introduced our proposed method which has accuracy of an instruction set simulator without sacrificing speed.

Chapter 6: Conclusion and Future Work

6.1. Conclusion

In today's network, data rates are increasing, protocols are becoming more dynamic and sophisticated and traffic for video and data applications is expected to grow exponentially. These challenges cannot be properly addressed with the rigid solutions provided by today's networking equipment. Networks need the ability to respond to changes, faults and scale performance to handle large volumes of client requests without creating unwanted delay.

To capitalize on this condition, telecom operators need to have more flexible, scalable and energy efficient processors, specifically designed for packet processing. One of the primary challenges in the design of network components, such as network processing units is to determine the best hardware architecture to support diverse applications.

We presented a flexible accurate host-compiled simulator to make it possible to explore wide ranges of architectures and application scenarios to find the optimal configuration that meets given performance, throughput and latency for programmable forwarding elements. In order to provide accurate system performance measurements, e.g. power consumption,

throughput rates, packet loss, latency statistics, or QoS requirements the framework supports back annotating host compiled simulation from an integrated OVP framework inside SystemC. The resulting approach offers faster and easier software development while enabling accurate analysis thanks to the integrated instruction-accurate simulator. This thesis advocates that application developers can use the simulator as a virtual prototype to simulate and debug their applications, executing it onto a given CPU architecture, using the integrated SystemC/OVP simulation. Then, forwarding device architects can use simulator to evaluate the trade-offs between different hardware/software design decisions in a reasonable simulation speed by using the proposed back-annotation method.

6.2. Future Work

Based on the work that has been done in this thesis and the obtained results, in the following some of the potential areas of study and suggestions for future work and research directions are presented:

6.2.1. Accurate Power Consumption Estimation

Increasing demand for more powerful embedded systems requiring smaller products with greater functionality and performance. Traditionally the main concern of a designer was performance and silicon area of the final product. Recently, power consumption has become a critical factor in embedded system design.

However, creating faster, smaller and low power products have conflicting requirements. On one hand, these devices require very low-power consumption, and on the other hand they require high performance for computational tasks, which leads to higher power consumption. For this reason, exploration of the large design spaces which depends upon the ability to assess each

design quickly, facilitates the efficient iteration over numerous candidate designs, shortens the time-to-market and therefore increases profits.

The simulation has been provided with useful tools (observers) to give the user a better sense of the risks of high power consuming designs. Observers can capture states of the processing elements, with their corresponding power consumption values, and all possible transitions, which will be later used to calculate power consumption. Observers are automatically registered in the system, so that any transition is automatically reported to the power logger. This relieves the responsibility of reporting changes in power consumption from the programmer, which must only define the state machine and trigger the transitions. Activities are completely user-defined. The simulator just registers them so that the information about current activities are attached to every power report.

Our approach is motivated by the observation in section 4.3.4 that under low incoming traffic rates, processing elements (threads inside cores) in network processor are idle for most of the time, leading to underutilization of available network resources on many workloads. If some of the processing elements are idle, it means that there is more processing power than required by the incoming packets. Different techniques, more detailed CPU models can be used for detecting idleness of threads to reduce power consumption. This presents an opportunity to turn off cores with little or no impact on the latency in the rest of the network.

Power consumption of an embedded application is a complex function of several components such as processing elements, memories, etc. Memory subsystem is a major energy consumer in several environments (e.g. server systems), hence, considering memory access power cost and providing methods to reduce the power consumption of the data memory can be very beneficial.

6.2.2. Back-annotation Using Prediction

No matter how efficient a simulator can be, the ever growing input rate and increasing sizes of routing tables make sure that we cannot simulate the network elements with real-world inputs and environment variables, and at some point we have to predict the parameters based on the datasets that we can actually simulate.

Efforts towards building accurate prediction models fall into two main directions: one is manually designing new features or new combinations of features to represent parameters that contribute the most to simulation timing; the other is using machine learning algorithms such as Support Vector Machine (SVM), Naive Bayes (NB). By doing so, a learning system can be trained, which can be used to back-annotate the host compiled model without the need to run the time consuming instruction set simulator.

Artificial neural networks (ANNs) are machine learning models that automatically learn to predict targets from a set of inputs. They have been used in research and commercially to guide autonomous vehicles [30], to play backgammon [36] and to predict weather [23], stock prices, and medical outcomes.

6.2.3. Automated Design Space Exploration Framework

As have shown earlier, one of the proposed simulator strength is enabling the investigation of architectural elements trade-offs. By designing an Automated Design Space Exploration framework, simulator is able to generate all possible designs and helps designer to choose the best design prior implementation based on different measures (speed, packet loss, throughput power consumption, etc.).

Appendix

1. Observers

IObserver.h

```
#ifndef IOBSERVER_H_
#define IOBSERVER_H_

class IObserver
{
public:
        /**
         * Function called by the NPU when a counter is added to a module
         * @param module_name         Module name to which counter was added
         * @param counter_name        Name of the counter
         * @param simulation time     Simulation time at which counter was added
         (defaults to 0 since counters are typically added before simulation begins)
         */
        virtual void counter_added(const std::string& module_name,
                  const std::string& counter_name, double simulation_time = 0) = 0;

        /**
         * Function called by the NPU when a counter is removed from a module
         * @param module name         Module name from which counter was removed
         * @param counter_name        Name of the counter
         * @param simulation_time     Simulation time at which counter was removed
         (defaults to 0 since counters are typically removed before simulation begins)
         */
        virtual void counter_removed(const std::string& module_name,
                                const std::string& counter_name, double simulation_time = 0) = 0;

        /**
         * Function called by the NPU when a counter is updated
         * @param module name         Module containing updated counter
         * @param counter_name        Name of the counter
         * @param new value           Current value of the counter
         * @param simulation_time     Simulation time at which counter value was updated
         */
        virtual void counter_updated(const std::string& module_name,
                                const std::string& counter_name, std::size_t new_value, double
                                simulation_time) = 0;


        /**
```

```
         * Function called by the NPU when data is written by a module
         * @param from_module         Module name of the transmitting module
         * @param data                JSON representation of data
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void data_written(const std::string& from_module,
                 const std::string& data, double simulation_time) = 0;

        /**
         * Function called by the NPU when data is read by a module
         * @param to_module           Module name of the receiving module
         * @param data                JSON representation of data
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void data_read(const std::string& to_module,
                const std::string& data, double simulation_time) = 0;

        /**
         * Function called by the NPU when data is dropped in a module
         * @param in_module           Module name in which data was dropped
         * @param data                JSON representation of data
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void data_dropped(const std::string& in_module,
                const std::string& data, const std::string& drop_reason, double simulation_time)=
                0;

        /**
         * Function called by the NPU when a processing element (pe) thread begins
         * @param pe_mod              PE in which thread was started
         * @param cluster_mod         Cluster containing the PE in which the thread was  started
         * @param thread_id           ID number of the thread
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void thread_begin(const std::string& pe_mod,
                const std::string& cluster_mod, std::size_t thread_id, std::size_t packet_id,
                double simulation_time) = 0;

        /**
         * Function called by the NPU when a processing element (pe) thread ends
         * @param pe_mod              PE in which thread was ended
         * @param clsuter_mod         Cluster containing the PE in which the thread was ended
         * @param thread_id           ID number of the thread
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void thread_end(const std::string& pe_mod,
                const std::string& cluster_mod, std::size_t thread_id, std::size_t packet_id,
                double simulation_time) = 0;

        /**
         * Function called by the NPU when a processing element (pe) thread starts idling
         * @param pe mod              PE in which thread is idling
         * @param clsuter_mod         Cluster containing the PE in which the thread is idling
         * @param thread_id           ID number of the thread
         * @param simulation_time     Simulation time at which event occurred
         */
        virtual void thread_idle(const std::string& pe_mod,
                const std::string& cluster_mod, std::size_t thread_id, std::size_t packet_id,
                double simulation_time) = 0;

        protected:
        /**
         * Default destructor
         */
        virtual ~IObserver() = default;
};

#endif /* IOBSERVER_H_ */
```

ThreadObserver.cpp

```
#include "ThreadObserver.h"
#include <iostream>

ThreadObserver::ThreadObserver(NPU &npu)
{
	npu.attach_observer(this);
	output_busy_ = new std::ofstream("busy_time.csv", std::ios::out);
	output_idle_ = new std::ofstream("idle_time.csv", std::ios::out);
}

ThreadObserver::~ThreadObserver()
{
	output_busy_->close();
	delete output_busy_;

	output_idle_->close();
	delete output_idle_;
}

void ThreadObserver::thread_begin(const std::string& pe_mod,
	const std::string& cluster_mod, std::size_t thread_id,
	std::size_t packet_id, double simulation_time)
{
	std::string thread_identifier = cluster_mod + " " + pe_mod + " " +
	std::to_string(thread_id);

	if(idle_.find(thread_identifier) != idle_.end())
	{
		double thread_idle_time = simulation_time - idle_[thread_identifier].second;
		*output_idle_ << cluster_mod << "," << pe_mod << "," << thread_id << "," <<
		thread_idle_time << std::endl;

		std::lock_guard<std::mutex> lock(idle_mtx_);
		idle_.erase(thread_identifier);
	}

	std::lock_guard<std::mutex> lock(begin_mtx_);
	begin_[thread_identifier] = std::make_pair(packet_id, simulation_time);
}

void ThreadObserver::thread_end(const std::string& pe_mod,
	const std::string& cluster_mod, std::size_t thread_id,
	std::size_t packet_id, double simulation_time)
{
	std::string thread_identifier = cluster_mod + " " + pe_mod + " " +
	std::to_string(thread_id);

	if(begin_.find(thread_identifier) != begin_.end())
	{
		double thread_busy_time = simulation_time - begin_[thread_identifier].second;
		*output_busy_ << cluster_mod << "," << pe_mod << "," << thread_id << "," <<
		thread_busy_time << std::endl;

		std::lock_guard<std::mutex> lock(begin_mtx_);
		begin_.erase(thread_identifier);
	}
}

void ThreadObserver::thread_idle(const std::string& pe_mod,
	const std::string& cluster_mod, std::size_t thread_id,
	std::size_t packet_id, double simulation_time)
{
	std::string thread_identifier = cluster_mod + " " + pe_mod + " " +
	std::to_string(thread_id);

	std::lock_guard<std::mutex> lock(idle_mtx_);
	idle_[thread_identifier] = std::make_pair(packet_id, simulation_time);
}
```

```
std::string ThreadObserver::make_result(const std::string& cluster_mod,
        const std::string& pe_mod, std::size_t thread_id) const
{
        return std::move(cluster_mod + "___" + pe_mod + "___" + std::to_string(thread_id));
}

std::string ThreadObserver::make_cresult(const std::string& cluster_mod,
        const std::string& pe_mod) const
{
        return std::move(cluster_mod + "___" + pe_mod);
}
```

NPU.CPP

```
#include "NPU.h"

SC_HAS_PROCESS(NPU);

NPU::NPU(sc_module_name nm)
: sc_module(nm), NPUModule(convert_to_string(nm))
{
        //...

        // Attach submodule interfaces
        // for the observers
        attach_interface(Parser->module_name(), Parser.get());
        attach_interface(Scheduler->module_name(), Scheduler.get());
        //...

        SC_THREAD(notify_observers);
}


void NPU::notify_observers()
{
        while(1)
        {
                auto func = events_.pop();
                func();
        }
}

void NPU::attach_observer(IObserver* observer)
{
        for(auto& each_module : common_interface_)
        {
                each_module.second->attach_observer(observer);
        }
        observers_.push_back(observer);
}

void NPU::attach_interface(const std::string& module_name, NPUModule* npu_module)
{
        common_interface_[module_name] = npu_module;
}
```

2. Timing Estimation with ISS integration

Baremetal.h

```
#ifndef BareMetalArmCortexASingle_TLM2_0_H_
#define BareMetalArmCortexASingle_TLM2_0_H_

#include "tlm.h"
#include "ovpworld.org/modelSupport/tlmPlatform/1.0/tlm2.0/tlmPlatform.hpp"
#include "ovpworld.org/modelSupport/tlmDecoder/1.0/tlm2.0/tlmDecoder.hpp"
#include "ovpworld.org/memory/ram/1.0/tlm2.0/tlmMemory.hpp"
#include "arm.ovpworld.org/processor/arm/1.0/tlm2.0/processor.igen.hpp"
////////////////////////////////////////////////////////////////////////////////
//                      BareMetalArmCortexASingle_TLM2_0 Class                //
////////////////////////////////////////////////////////////////////////////////
class BareMetalArmCortexASingle_TLM2_0 : public sc_core::sc_module {

public:
  BareMetalArmCortexASingle_TLM2_0 (sc_core::sc_module_name name, const char *variant);

  icmTLMPlatform      Platform;
  decoder             <2,2> bus1;
  ram                 program;
  ram                 stack;
  arm                 cpu1;

  icmAttrListObject *attrsForcpu(const char *variant) {
    icmAttrListObject *userAttrs = new icmAttrListObject;
    userAttrs->addAttr("endian",        "little");
    userAttrs->addAttr("compatibility", "nopSVC");
    userAttrs->addAttr("variant",       variant);
    userAttrs->addAttr("UAL",           "1");
    return userAttrs;
  }
}; /* BareMetalArmCortexASingle_TLM2_0 */


#endif /*BareMetalArmCortexASingle_TLM2_0_*/
```

BareMetal.cpp

```
#include "BareMetalArmCortexASingle_TLM2_0.h"

BareMetalArmCortexASingle_TLM2_0::BareMetalArmCortexASingle_TLM2_0 (
sc_core::sc_module_name name, const char *variant)
: sc_core::sc_module (name)
, Platform ("icm", ICM_VERBOSE | ICM_STOP_ON_CTRLC| ICM_ENABLE_IMPERAS_INTERCEPTS )
, bus1("bus1")
, program ("program", "sp1", 0x100000)
, stack    ("stack",   "sp1", 0x100000)
, cpu1 ( "cpu1", 0, ICM_ATTR_DEFAULT, attrsForcpu(variant))

{
  // // bus1 masters
  cpu1.INSTRUCTION.socket(bus1.target_socket[0]);
  cpu1.DATA.socket(bus1.target_socket[1]);

  // bus1 slaves
  bus1.initiator_socket[0](program.sp1);     // Program Memory
  bus1.setDecode(0, 0x00000000, 0x000fffff);

  bus1.initiator_socket[1](stack.sp1);       // Stack Memory
  bus1.setDecode(1, 0x00100000, 0x001fffff);

}
```

Processor.h

```
#ifndef PROCESSOR_H_
#define PROCESSOR_H_
#include <utility>
#include "../BareMetalArmCortexASingle_TLM2_0.h"

class BareMetalArmCortexASingle_TLM2_0;
class Processor: public ProcessorSIM {
public:
  SC_HAS_PROCESS(Processor);
  /*Constructor*/
  explicit Processor(sc_module_name nm, std::string configfile = "");
  /*Destructor*/
  virtual ~Processor() = default;

public:
  void init();
  void process();
  BareMetalArmCortexASingle_TLM2_0 top;

public:
  void Processor_PortServiceThread();
  void ProcessorThread(std::size_t thread_id);
  std::vector<sc_process_handle> ThreadHandles;
  std::queue<std::shared_ptr<PHV>> jobs;
  std::queue<std::shared_ptr<CPAMessages>> tableupdates;
  sc_event GotaJob, GotATableUpdate;

};

#endif  // PROCESSOR_H_
```

Processor.cpp

```
#include "Processor.h"
#include <string>
#include "InternalObject.h"

Processor::Processor(sc_module_name nm, std::string configfile ): top("top", "Cortex-A9UP")
{
    if( access( constants::TIMING_FILE, F_OK ) != -1 )
    {
        remove(constants::TIMING_FILE);
    }

    ThreadHandles.push_back( sc_spawn(
        sc_bind(&Processor::Processor_PortServiceThread, this)));
}

static ICM_MEM_WATCH_FN(start_flag_readCallBack)
{

    //write pkt to OVP CPU memory
    sc_process_handle this_process = sc_get_current_process_handle();
    sc_object* current_scmodule = this_process.get_parent_object();
    sc_object* proc = current_scmodule->get_parent_object()->get_parent_object();
    string module_name_ = "Processor Read Callback";

    if(Processor* p = dynamic_cast<Processor*>(proc))
    {
       if (p->jobs.size() != 0)
       {
               auto job = p->jobs.front();
               p->jobs.pop();
               int rec = 0;
               uint32_t srcip = 0;
               uint32_t dstip = 0;
```

```
cout<<"@"<<sc_time_stamp()
<<" Read Call Back: from input port read: "
<< job->id() << std::endl;

p->buffer.push(job);
rec = job->id();
srcip = ntohl(job->parsed_hdr->getIPSrc().s_addr);
dstip = job->parsed_hdr->getIPDst().s_addr;

// create an internal object to pass to OVP CPU
InternalObject intObj(CommandType::process, srcip, 0, dstip, job->id());

icmWriteProcessorMemory(processor,address,&intObj,sizeof(intObj));
return;

}
else if (p->tableupdates.size() != 0)
{
        auto request = p->tableupdates.front(); //CPAMessages
        p->tableupdates.pop();
        npulog(normal, std::cout << "Update OCCURED" << std::endl;)

        if (request->match_key.type == MatchKeyParam::Type::LPM)
        {
                npulog(normal,std::cout << "Prefix to update is --LPM " <<
                std::endl;)

                InternalObject intObj(CommandType::update,
                        ntohl(to_int(request->match_key.key)),
                        request->match_key.prefix_length,
                        to_int(request->action_data),
                        p->lpm_update_counter++,
                        request->match_key.type);
                icmWriteProcessorMemory(processor,address,&intObj,
                sizeof(intObj));
                return;

        }
        else if(request->match_key.type == MatchKeyParam::Type::EXACT)
        {
                npulog(normal,std::cout <<
                "Prefix to update is --EXACT" << std::endl;)
                struct in_addr in;
                in.s_addr = to_int(request->match_key.key);


                InternalObject intObj(CommandType::update,
                        ntohl(to_int(request->match_key.key)),
                        0,
                        0,
                        p->exact_update_counter++,
                        request->match_key.type);

                        std::copy(request->action_data.begin()
                        ,request->action_data.end(),
                        intObj.ExactActionValue);

                        icmWriteProcessorMemory(processor,address,&intObj
                        ,sizeof(intObj));
                        return;
        }
}
else
{
        // No Jobs request a job from the schedular
        auto JobRequest = std::make_shared<SchedularMessages>
        (0, CommandType::JobRequest);
        p->out[1]->put(JobRequest);
        npulog(normal, std::cout <<
        "Requesting a Job right now" << std::endl;)
        wait(p->GotaJob | p->GotATableUpdate);
```

```
            }
    }
    else
    {
             npu_error("Processor Handle- Cast failed");
    }
}

static ICM_MEM_WATCH_FN(stop_flag_writeCallBack)
{
    string module_name_ = "Processor Write Callback";

    InternalObject res;
    icmReadProcessorMemory(processor,thing, &res, sizeof(res));

    ofstream timing(constants::TIMING_FILE, ios::app);
    if(!timing.is_open())
    {
        std::cout << "the file could not opened" << std::endl;
        return;
    }

    unsigned instCount =0;
    icmReadProcessorMemory(processor,constants::FIRST, &instCount, sizeof(instCount));
    unsigned instCount2 =0;
    icmReadProcessorMemory(processor,constants::SECOND, &instCount2,
       sizeof(instCount2));

    //write to file
    timing <<res._cmd<<", address: "<< constants::FIRST<<", "
       << MatchKeyParam::type_to_string(res.type)<<", "<<instCount<<"," <<instCount2<<endl;

    timing.close();
    //write to file end

    sc_process_handle this_process = sc_get_current_process_handle();
    sc_object* current_scmodule = this_process.get_parent_object();
    sc_object* proc = current_scmodule->get_parent_object()->get_parent_object();
    if(Processor* p = dynamic_cast<Processor*>(proc))
    {
        if (p->buffer.size() != 0)
        {
            auto sendpkt = p->buffer.front();
            p->buffer.pop();

            sendpkt->parsed_hdr->ip->ip_dst.s_addr = res.LPMActionValue;

            std::copy(res.ExactActionValue,
                res.ExactActionValue+ETHER_ADDR_LEN,
                sendpkt->parsed_hdr->ethernet->ether_dhost);

                wait((instCount+instCount2)/1000,SC_MS);   //1 MIPS

                p->out[0]->put(sendpkt);
                cout<<"@"<<sc_time_stamp()<<" Write Call Back: sent:  ";
                sendpkt->parsed_hdr->dumpipdst();
        }

    }
    else
    {
            cout<<"@"<<sc_time_stamp()<<" Write Call Back: Casting Failed!!!"
               << std::endl;
            assert(false);
    }

}


void Processor::Processor_PortServiceThread()
{
```

```
    // Thread function to service input ports.
     //running flag
    top.cpu1.addReadCallback(running_flag, running_flag, &start_flag_readCallBack);
    //application done processing flag
    top.cpu1.addWriteCallback(stop_flag, stop_flag,  &stop_flag_writeCallBack);

    top.cpu1.loadLocalMemory("myapp.ARM_CORTEX_A9.elf",
         (icmLoaderAttrs)(ICM_LOAD_VERBOSE | ICM_SET_START));

    while (1)
    {
        auto recevied_tr = in->get();
        if (auto header = std::dynamic_pointer_cast<PHV>(recevied_tr)) {
            jobs.push(header);
            GotaJob.notify();
        } else if (auto cpreqs = std::dynamic_pointer_cast<CPAMessages>(recevied_tr)) {
            tableupdates.push(cpreqs);
            GotATableUpdate.notify();
        } else {
            npu_error("Processor - All casts failed");
        }
    }
}
```

Application.cpp

```
#include <stdio.h>
#include <iostream>
#include "InternalObject.h"


class TrieManager {
public:
  TrieManager():trie(2,sizeof(0)){}
  ~TrieManager()=default;
  PrefixTree<uint32_t> trie ;
};

class ExactTrieManager {
public:
  typedef std::vector<u_char> actiontype;
  ExactTrieManager():trie({0,0,0,0,0,0}, sizeof(defaultaction)){}
  ~ExactTrieManager()=default;
  HashTrie<actiontype> trie;
  actiontype defaultaction;
};

int main()
{
  run = true;

  printf("Application starting...\n");
  TrieManager matchactiontables;
  ExactTrieManager exactmatchtables;


  while(run)
  {
    unsigned int octet[4]  = {0,0,0,0};
    if(pkt._cmd == CommandType::update)
    {


      if(pkt.type == MatchKeyParam::Type::LPM) {
        std::cout << "Application Update LPM: "<<pkt.id << std::endl;
        convertToIp(octet, pkt.LPMActionValue);

        int temp = impProcessorInstructionCount();
        BitString updateentry(pkt.MatchValue,pkt.prefixlength);
        RoutingTableEntry<uint32_t> lpmupdate
               (updateentry,sizeof(updateentry),pkt.LPMActionValue,sizeof(pkt.LPMActionValue));
```

```
        RoutingTableEntry<uint32_t>* Entry = &lpmupdate;
        matchactiontables.trie.update(Entry,1,Trie<uint32_t>::Action::Add);
        t_adr = impProcessorInstructionCount() - temp;
        t_adr2 = 0;

      } else if (pkt.type == MatchKeyParam::Type::EXACT) {

        std::cout << "Application Update EXACT: "<<pkt.id << std::endl;

        int temp = impProcessorInstructionCount();
        BitString updateentry(pkt.MatchValue);
        ExactTrieManager::actiontype action(pkt.ExactActionValue,pkt.ExactActionValue+6);

        RoutingTableEntry<ExactTrieManager::actiontype> epmupdate
        (updateentry,sizeof(updateentry),action,sizeof(action));
        RoutingTableEntry<ExactTrieManager::actiontype>* Entry = &epmupdate;

        exactmatchtables.trie.update(Entry, 1, Trie<ExactTrieManager::actiontype>::Action::Add);
        t_adr2 = impProcessorInstructionCount() - temp;
        t_adr = 0;

      } else {
        std::cout << "Invalid Match Key Type" << std::endl;
      }

    }
    else if (pkt._cmd  == CommandType::process)
    {
      BitString ip(pkt.MatchValue);
      int temp = impProcessorInstructionCount();
      auto result = matchactiontables.trie.longestPrefixMatch(ip);
      t_adr = impProcessorInstructionCount() - temp;
      pkt.LPMActionValue = result;

      temp = impProcessorInstructionCount();
      auto resultemp = exactmatchtables.trie.exactPrefixMatch(ip);
      t_adr2 = impProcessorInstructionCount() - temp;
      std::copy(resultemp.begin(),resultemp.end(),pkt.ExactActionValue);
    } else {
      std::cerr << "Unknown CommandType" << std::endl;
      impFinish();
    }
    stop = true;

  }

  printf("finishing...\n");

  impFinish();

}
```

Bibliography

[1] M. A. Franklin, P. Crowley, H. Hadimioglu, and P. Z. Onufryk, *Network Processor Design: issues and practices*: Morgan Kaufmann, 2003.

[2] J. C. Maxwell, *A treatise on electricity and magnetism*: Clarendon press, 1881.

[3] C.-K. Lo, L.-C. Chen, M.-H. Wu, and R.-S. Tsay, "Cycle-count-accurate processor modeling for fast and accurate system-level simulation." pp. 1-6.

[4] S. Pasricha, and N. Dutt, *On-chip communication architectures: system on chip interconnect*: Morgan Kaufmann, 2010.

[5] K. P. Lawton, "Bochs: A portable pc emulator for unix/x," *Linux Journal,* vol. 1996, no. 29es, pp. 7, 1996.

[6] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator." pp. 23-34.

[7] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, and N. Binkert, "Asim: A performance model framework," *Computer,* vol. 35, no. 2, pp. 68-76, 2002.

[8] L. B. R. Bhargava, and B. Sander, " "Amd," personal email communication."

[9] D. Chiou, H. Angepat, N. A. Patil, and D. Sunwoo, "Accurate functional-first multicore simulators," *IEEE Computer Architecture Letters*, no. 2, pp. 64-67, 2009.

[10] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling." pp. 74-85.

[11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti, "The gem5 simulator," *ACM SIGARCH Computer Architecture News,* vol. 39, no. 2, pp. 1-7, 2011.

[12] N. Hardavellas, S. Somogyi, T. F. Wenisch, R. E. Wunderlich, S. Chen, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Simflex: A fast, accurate, flexible full-system simulation framework for performance evaluation of server architecture," *ACM SIGMETRICS Performance Evaluation Review,* vol. 31, no. 4, pp. 31-34, 2004.

[13] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *Computer,* vol. 35, no. 2, pp. 59-67, 2002.

[14] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald III, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores." pp. 1-12.

[15] Y. Luo, J. Yang, L. N. Bhuyan, and L. Zhao, "NePSim: A network processor simulator with a power evaluation framework," *Micro, IEEE,* vol. 24, no. 5, pp. 34-44, 2004.

[16] P. K. Szwed, D. Marques, R. M. Buels, S. McKee, and M. Schulz, "SimSnap: Fast-forwarding via native execution and application-level checkpointing." pp. 65-74.

[17] A. Sandberg, E. Hagersten, and D. Black-Schaffer, "Full Speed Ahead: Detailed Architectural Simulation at Near-Native Speed," 2014.

[18] "Standard Performance Evaluation Corporation," http://www.spec.org/cpu2000/

[19] J. Teich, “Hardware/software codesign: The past, the present, and predicting the future,” *Proceedings of the IEEE,* vol. 100, no. Special Centennial Issue, pp. 1411-1430, 2012.
[20] D. J. Lilja, “MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research,” *Computer Architecture Letters,* vol. 1, no. 1, pp. 7-7, 2002.
[21] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” *ACM SIGOPS Operating Systems Review,* vol. 36, no. 5, pp. 45-57, 2002.
[22] A. Blankstein, S. Erickson, and M. Melara, “Mininet Clustering.”
[23] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks." p. 19.
[24] M. Gupta, J. Sommers, and P. Barford, "Fast, accurate simulation for SDN prototyping." pp. 31-36.
[25] U. A. Samar Abdi, Gordon Bailey, Bochra Boughzala, Faras Dewal, Shafigh Parsazad, Eric Tremblay, “PFPSim: A Programmable Forwarding Plane Simulator,” in ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS) 2016, In press.
[26] T. Grtker, S. Liao, G. Martin, and S. Swan, “System Design with SystemC,” 2010.
[27] D. Pursley, *How the Productivity Advantages of High-Level Synthesis Can Improve IP Design, Verification, and Reuse*, 2013.
[28] J. B. Gosling, *Simulation in the design of digital electronic systems*: Cambridge University Press, 1993.
[29] T. Bouhadiba, M. Moy, F. Maraninchi, J. Cornet, L. Maillet-Contoz, and I. Materic, "Co-simulation of functional SystemC TLM models with power/thermal solvers." pp. 2176-2181.
[30] J. Manner, “Performance evaluation of software switching using commodity hardware,” Aalto University, 2012.
[31] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, high performance ethernet forwarding with cuckooswitch." pp. 97-108.
[32] D. Intel, “Data Plane Development Kit,” *URL http://dpdk. org*.
[33] R. Giladi, *Network processors: architecture, programming, and implementation*: Morgan Kaufmann, 2008.
[34] "Ericsson SNP 4000 Smart Network Processor," http://www.ericsson.com/ourportfolio/products/ssr-8000-family.
[35] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, “A survey of software-defined networking: Past, present, and future of programmable networks,” *Communications Surveys & Tutorials, IEEE,* vol. 16, no. 3, pp. 1617-1634, 2014.
[36] I. Prívara, B. Rovan, P. Ružička, and P. Ruzicka, *Mathematical Foundations of Computer Science 1994: 19th International Symposium, MFCS'94, Kosice, Slovakia, August 22-26, 1994. Proceedings*: Springer Science & Business Media, 1994.
[37] W. J. Dally, and B. Towles, "Route packets, not wires: on-chip interconnection networks." pp. 684-689.
[38] S. D. Chawade, M. A. Gaikwad, and R. M. Patrikar, “Review of XY routing algorithm for Network-on-Chip architecture,” *International Journal of Computer Applications,* vol. 43, no. 21, pp. 975, 2012.

[39] J. Sommers, H. Kim, and P. Barford, "Harpoon: a flow-level traffic generator for router and network tests." pp. 392-392.
[40] "P4.org," http://p4.org/.
[41] "The LLVM Project, “Writing an LLVM Pass — LLVM 3.6 documentation,” The LLVM Compiler Infrastructure.," 30 Nov, 2015; http://llvm.org/docs/WritingAnLLVMPass.html#writing-an-llvm-pass-functionpass.
[42] G. Varghese, *Network algorithmics*: Chapman & Hall/CRC, 2010.
[43] "OpenRISC 1200 Processor.," http://opencores.org/or1k/OR1200_OpenRISC_Processor.
[44] "OpenCore.," http://opencores.org/.
[45] "Embedded software development using an interpretive instruction set simulator," http://www.design-reuse.com/articles/21745/interpretive-instruction-set-simulator.html.
[46] "Open Virtual Platforms," http://www.ovpworld.org.
[47] T.-c. Chiueh, and P. Pradhan, "High-performance IP routing table lookup using CPU caching." pp. 1421-1428.
[48] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Communications Magazine,* vol. 53, no. 2, pp. 90-97, 2015.
[49] "Cisco Blogs," http://blogs.cisco.com/sp/global-internet-routing-table-reaches-512k-milestone.