SYSTEM-LEVEL MODELING OF PROGRAMMABLE PACKET PROCESSING SYSTEMS

UMAIR AFTAB

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: Umair Aftab

Entitled: "System-level Modeling of Programmable Packet Processing Systems"

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 the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair

_____ Examiner

_____ Examiner

_____ Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____
Dean of Faculty

Date _____

ABSTRACT

Computer networks are experiencing explosive growth which is reinforced by the recent exhaustion of the global IPv4 addresses space in 2011 and the tenfold increase in users from 1999 to 2013. The advent of cloud, mobile and IoT is only going to accelerate this growth. This accedes the need for flexible and scalable networks that process packets faster. Programmable packet processing systems have emerged as a solution which aim to find balance between flexibility of supporting different processing functions while maintaining a high processing capability. Designing architectures that support such paradigms is fairly complicated as decisions need to be made for evaluating trade-offs between flexibility and efficiency. Questions like what programmatic interfaces, services, applications and protocols are required need to be answered before synthesis of actual hardware. To evaluate such requirements modelling techniques are required to evaluate architecture decisions accurately early enough in the design phase.

In this thesis, we propose a flexible system level modelling methodology for early validation, design and analysis of packet processing applications for programmable forwarding plane architectures. The hardware and software architecture is described in a high level language which can be used to describe forwarding planes from many core network processors to reconfigurable processing pipelines. Device architects can use this for design space exploration, prototyping and validation; where application developers can start pre-silicon application design, development and debugging to evaluate different hardware and software decisions in an industry with ever shrinking market windows.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Dr. Samar Abdi for guiding me throughout my research work. My work under his guidance has not only improved my technical and presentation skills but has improved my grasp of system level design and modelling approaches as a whole. I highly appreciate the open minded view and clear-sighted objective guidance towards new ideas and opportunities, for which I am greatly thankful.

I would like to thank the work of my project team members Gordon Bailey, Shafigh Parsazad, Eric Trembley, Kamil Saigol and Faras Dewal without whom all this would not have been possible. I would also like to acknowledge Ericsson Research Canada and NSERC for funding this research project and providing with the required industry direction and technical expertise.

Finally I would like to thank my parents for all of their support.

EX NIHILO NIHIL FIT

Contents

List of Figures

List of Tables

Code Listings

Chapter 1: Introduction

Computer networks are growing at an unprecedented rate with the addition of smartphone and handheld devices. With the advent of the cloud most applications are being pushed online to datacenters. The distinction between what is mobile content and what is not is being increasing blurred. Content creators pushing towards a more unified experience towards all devices fixed or mobile. To keep up with this ISPs, Mobile network operators and datacenter designers need to design networks that are no longer static but instead can scale, react and be managed easily.

Networks deploy a slew of devices from switches and routers to firewalls, load balances and intrusion detection boxes. Historically to monitor and configure, network vendors have had to solely rely on CLI or SNMP agents. CLI due to its inherent nature requires configuration to be done individually for every device in the network. SNMP by design is iterative and inefficient, parallel connections need to be polled since devices do not have the ability to subscribe to configuration streams over TCP or report back failures. Although some proprietary network management tools offer a central point for configuration, these tools still function on individual protocols and configuration interfaces. This mode of operation does not offer introspection into the operational state of the network, only a limited set of metrics can be collected to gauge the health of a network which again are highly device and vendor dependent since the implementation of protocols is often proprietary and run in black boxes. This increases network complexity, capital requirements and operational overhead for the operator to run the network.

This has given rise to rapid adoption of paradigms like Software Defined Networking (SDN) in recent years, by both network operators and several switch chip vendors offering support for the OpenFlow protocol [1] to implement SDN architectures. SDN abstracts and ramifies

network systems which make decisions of where the traffic should go from the underlying systems that actually route network traffic, these are referred to as the control plane and forwarding plane respectively. The question we need to ask is what problem does SDN solve? The customary answer is automating configuration management. It does so by assimilating network control of multiple operation points into a single software control program. This approach enables programmable network control because it decouples control of the network from forwarding functions allowing non-blocking asynchronous operation of routing and forwarding functions.

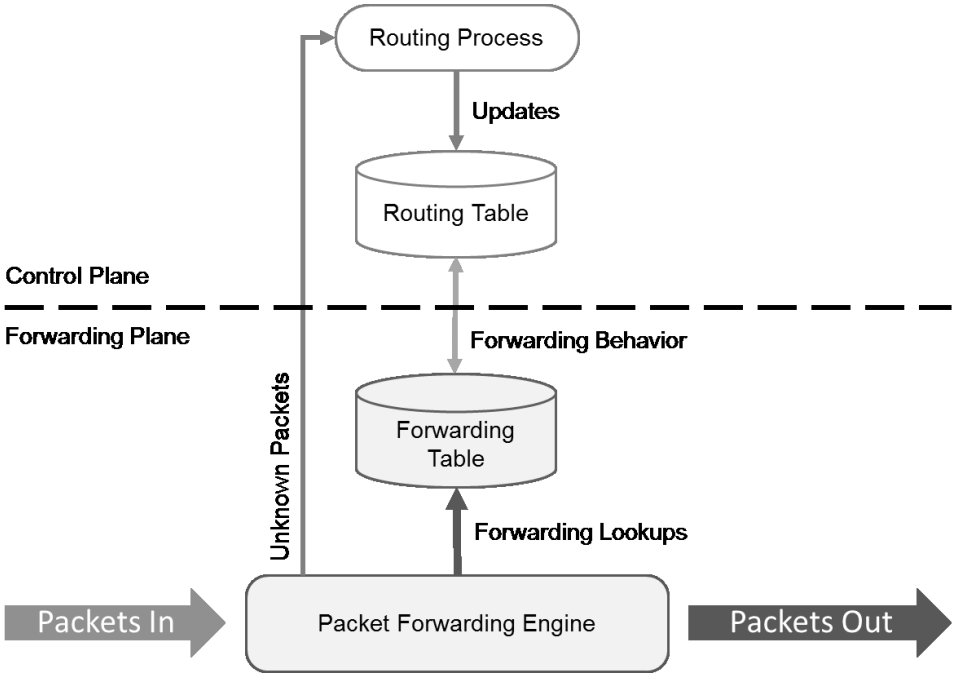

Figure 1 SDN Architecture

A Control Plane in a router makes the decision of where the traffic is sent. It configures, updates or modifies the routing tables for the data planes they manage according to the topology and metrics of the network by exchanging routing information with other routers over network discovery and routing protocols like BGP, RIP or OSPF.

Forwarding planes or data planes form part of the routing systems that actually handle network traffic. Packets are processed by using information from forwarding tables populated by the control plane to essentially figure out how to get the packet to its destination by forwarding it to the appropriate next hop in the network. The SDN prerogative is that the application executing on forwarding plane determines the type of device the forwarding plane represents.

A SDN switch (Control + Data plane) has one or more tables of rules for packet handling. Each rule matches a subset of the traffic which performs certain actions of a matching packet. Depending on the rules an SDN device can behave like a router, switch or a firewall or any combination of a network device. Designing architectures that support such paradigms is fairly complicated as decisions need to be made to evaluate trade-offs for programmable network elements that offer flexibility for new functional patterns while being scalable to support ever increasing network requirements. Questions like what programmatic interfaces, services, applications and protocols are required need to be answered before synthesis of actual hardware. To evaluate such requirements modelling techniques are required to evaluate architecture decisions accurately early enough in the design phase.

1.1. Packet Processing

Packet processing applications by nature exhibit a high degree of data parallelism. In theory, stateless processing can be parallelized indefinitely, given enough processing and memory resources, since each processing thread works independently on a given packet. Moreover, most of the processing actions are fairly simple, often a sequence of arithmetic operations. The complexity, typically, lies in matching an incoming packet header against a set of header patterns stored in a table. The match type can be exact, longest-possible (longest-prefix), range or wildcard.

Forwarding device architects exploit the data parallelism in packet processing applications in one of two ways: either by using a large number of multi-threaded RISC cores or a deep pipeline. The former template is used in Network Processing Units (NPUs), while the latter is used in reconfigurable hardware pipelines. In order to mitigate the complexity of header field matching, architects use efficient search-and lookup data structures such as tries, or hardware accelerators such as ternary content-addressable memories (TCAMs) that perform single cycle matches.

To increase throughput and support higher bandwidth for networks forwarding planes need to be blazingly fast. Several design factors affect performance of forwarding planes for packets. The critical path in any forwarding plane can be defined as:

1. Ingress link layer deserialization and extracting the packet.
2. Identifying and decoding packet headers.
3. Processing packet – performing field lookups.
4. Sending packet through forwarding plane fabric (fixed-function processing elements).
5. Serialization and egress data link encapsulation.

1.2. Motivation

Although SDN has seen rapid adoption among network operators and chip vendors by offering support for OpenFlow it is still however protocol dependent and is limited to known packet header types, this allows limited design space exploration for new efficient networking protocols and their performance. New proposals for protocol-independent programming abstractions like P4 have been proposed. These focus on describing how packets are processed regardless of the hardware implementation. This allows the support for different hardware while providing support for standard abstractions for programming forwarding planes.

The notion of a programmable forwarding plane is not new. Network processors (NPUs) and even general purpose CPUs have long been used in forwarding plane hardware [2], [3], [4]. Recently, we have seen the advent of new platforms, such as reconfigurable match table (RMT) [5] and FlexPipe [6] that utilize reconfigurable pipelined hardware for fast and programmable packet processing. Based on these trends, and the rapid pace at which networks are growing, network operators need to be able scale faster reacting to traffic demands, faults, and bad routing links. All the while optimizing on operation cost, performance and tolerance which determines the bottom line for the operator in a market whose time to market windows grow shorter and shorter with each financial quarter. This translates the need to develop faster flexible forwarding planes.

Shorter development cycles require co-development of software and hardware design cycles. Application developers want to develop, debug and analyze their packet processing programs even before the availability of silicon. System architects want to explore architectural options quickly. The design space for forwarding elements is multifaceted and diverse with each type of network application dictating its own set of workload and application requirements.

Having a model at the appropriate abstraction level provides a flexible path for design architects and application developers to narrow down the design space and explore optimal designs before deciding on an implementation. In order to support such an ecosystem, in this thesis we propose a simulation methodology to describe and generate host-compiled models of the hardware and low level software services. It reduces the modelling effort by describing the architecture in an abstract high-level language. The user can integrate packet processing applications like P4 by linking against this model. The simulation model can be traced to extract metrics like latency, drops, memory consumption, etc. This drives to establish and concretize the design requirements for the final implementation of the software stacks and the forwarding plane hardware.

1.3. Thesis Contribution

The main contributions of this thesis are presented as follows:

1.3.1. Memory Abstraction Layer for Host-Compiled Models

Host compiled simulations unlike instruction set simulators have the main limitation that all allocation and accesses of application happen directly on the host, although great for application development it does not give any indication of performance on the target platform, which can be observed in cycle-accurate simulators. Direct allocation allows host-compiled simulations to execute natively and not require the enormous modelling effort than their counterpart simulators. This however greatly limits the accuracy of the simulation for architectural evaluation and design trade-offs.

In this thesis we present a novel compile time solution that leverages the C++ access and allocation semantics and the C++ memory model to provide a base layer which is used to redirect memory allocation and access to memory models instead of directly on the host.

1.3.2. Forwarding Architecture Description

In the past the hardware design was primarily developed using VHDL and Verilog, as systems grow more and more complex, with increasing gate count for every generation and smaller process nodes, they present their own challenges in which the window to markets can no longer support years of chip development. Increasing faster development is vital to the design of modern electronic systems. The limitations and inexpressibility of Verilog and VHDL for software and algorithms gave rise to the development of concepts of TLM and SystemC in 2002. SystemC is C++ library with a discreet event simulation kernel. It provides all of the constructs of hardware

design languages like VHDL but allow the integration of normal C++ algorithms. This allows system level designers to model systems at higher abstraction levels for kick starting application development and evaluating architectural trade-offs in regards to system level design. Although SystemC provides an optimal integration of C++ and hardware modelling constructs it still befalls the trap of VHDL like languages that are too detailed of a specification language in which every detail of hardware has to be intricately described.

In this thesis we take the next step of building on top of the design approaches of SystemC and TLM to propose a faster approach of modelling systems which abstracts away a lot of the architectural specification details in a compact description to allow engineers to concentrate on the functionality of the system more than the implementation of the system in simulation.

1.3.3. Automatic Model Generation

This thesis presents the forwarding architecture description language, but like every other language by itself it is useless if nothing can be done with it. We describe the implementation of the FAD to SystemC compiler, which generates a SystemC C++11 model from the description. The compiler generates all of the SystemC code base for the modules and their interconnections, this reduces the modelling effort for the designer and allows more focus on the behavioural aspect of the modules and developing applications for the platform. Since the FAD description is compiled, quick structural changes can be made to it which the compiler takes care of updating the SystemC code base for the model.

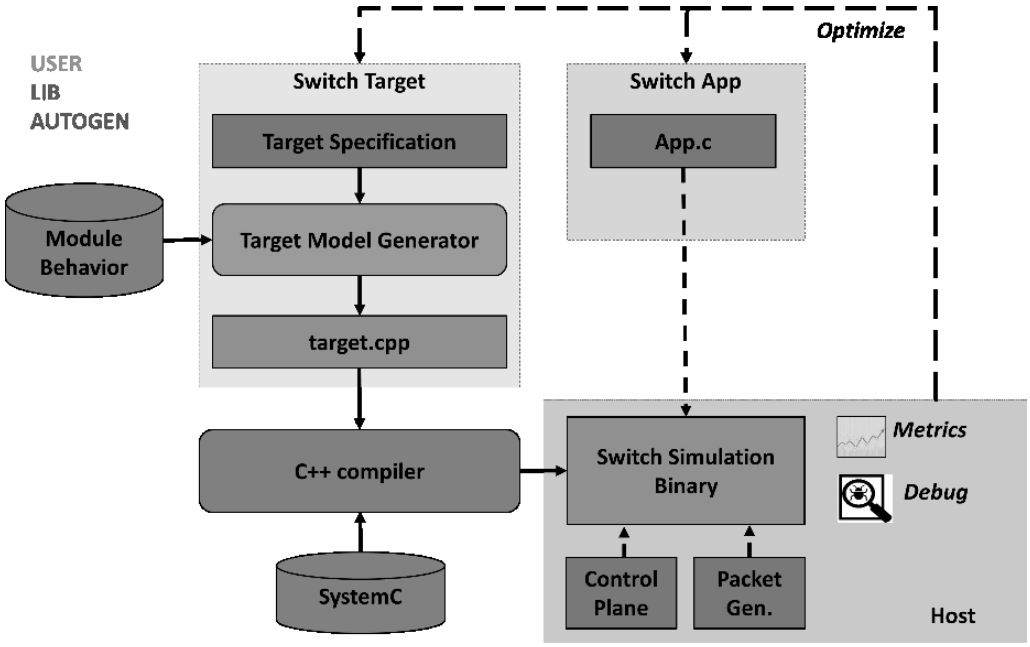

Figure 2 Modelling methodology

In addition to the compiler we provide a runtime library which has common utilities already available to the user for estimation, exploration and evaluation. The logic for the modules in the forwarding device is written in C++ by the user. The packet processing program to be executed by the programmable cores in the forwarding device is compiled and linked with the SystemC model of the forwarding device.

1.4. Related Work

Comparatively there has been little work on system level modeling of programmable forwarding planes, given that the majority of switches in the market are fixed function. NePSim is a cycle-level network processor simulator which models the Intel IXP1200 architecture [7]. Since it models a specific Intel network processor, this makes it unsuitable to adapt for architectural exploration. There are SDN simulators and emulators, such as Mininet [8] and fs-SDN [9] but they do not model the forwarding plane architecture and are more geared towards emulating networks on the whole; such simulators are therefore unsuitable for evaluating design trade-offs.

There are processor simulators like MCSimA+ [10], Gem5 [11] and OVPSim [12] which perform full system simulation of x86 processing cores. Gem5 and OVPSim support for a variety of processor platforms like x86, ARM and PowerPC. PTLSim is another full system x86 cycle accurate Athlon micro architectural simulator. It models a modern superscalar out of order x86-64 processor core unlike OVPSim which is only instruction accurate.

These simulators can be used to model performance of soft-switch implementations but have difficulty scaling due to the detailed nature of instruction/cycle-level simulation, they cannot however model complete architectures of packet processing systems. To the best of our knowledge, there is currently no other published simulator than PFPSIM [3] that enables system-level modeling and simulation of programmable forwarding devices.

Chapter 2: Programmable Forwarding Planes

The fundamental task of any forwarding element is to switch packets from its input to the appropriate output by modifying the appropriate headers of the packet according to its forwarding table. This can seem trivial but different applications have different requirements which dictate the trade-offs between complexity, speed and flexibility of implementations. Early switches were simple fixed function devices implemented as ASICs that switched packets at the L3/L2 level. Although ASIC implementations always afford us high-performance the trade-off is application flexibility and development cost.

As networks grow larger the design requirements change to accommodate more complex applications. The need for new efficient and flexible routing that scales stresses the development of new protocols and optimized algorithms in forwarding elements. Since most functionality in AISCS directly translates to silicon implementing new protocols and algorithms in most cases

requires respinning the whole chip which can take a long time to develop, synthesize and verify. Development cycles for ASICs fall in the range of years and cannot simply keep pace with rapid pace at which networks are evolving. This has given way to the development of distributed processing platforms like networks processors or deep pipelines like FlexPipe and RMT, each implementation gearing towards particular networking applications. This of course does not mean that ASICs are going to be fade out. As is with every industry they will get relegated to implementations where flexibility has to be sacrificed for applications with the most bleeding edge high performance requirements, e.g. Backbones and Regional Tier 1 networks.

2.1. Forwarding Plane Programming

OpenFlow was introduced in 2007 as a way for software control planes to manage data planes switches. It is a standard that allows the population of forwarding tables such as Ethernet hash tables, Longest prefix match tables for IP and Wildcard searches for access control lists (ACL) to name a few by abstracting away hardware implementations. The OpenFlow conjecture is that switches are fixed-function in the sense that they have well known rigid functions that stem from and can be performed on protocols which are ratified by IEEE and IETF. In the first version of OpenFlow tables only for Ethernet, IPv4, VLANs, and ACLs [13] could be populated; later versions added support for more protocol headers like IPv6.

These protocols are usually directly implemented in silicon by ASIC networking chip vendors. Although OpenFlow focuses more on the standardization of the control plane software stack, the software stack of forwarding planes is usually bare metal provided by chip vendors which just expose the hardware-accelerated functions and don’t allow for much application flexibility for introducing more efficient transport or routing protocols.

OpenFlow has been more focused on assimilating control planes and management than operation of data planes, with data planes having OpenFlow agents which implement the specification and remains to be vendor controlled. This is why OpenFlow historically has been restricted to known packet header types and repeated extensions to the specification have been made to support new protocols. This makes it cumbersome to implement or even support new headers and protocols as they must be both agreed upon by OpenFlow and the chip vendors. Therefore, researchers have proposed new programming abstractions, such as P4 and Intel DPDK, for the forwarding plane in order to enable programming the forwarding chip to support new protocols [14].

The Intel DPDK framework is a set of libraries to accelerate packet processing in soft switches by abstracting away hardware and software environments by providing an application programming interface to available hardware accelerators, OS networking stack and other hardware like PCI bus controllers and DMA engines. It also provides a software stack for lockless queues, pre-allocated pools, circular buffers and low overhead asynchronous polling drivers. Although initially for x86 and Itanium processors it has been ported to other architectures like IBM POWER8.

2.1.1. P4 - Protocol Independent Programming Abstraction

P4 aims to be a protocol independent programming abstraction by providing a standard API for developing applications which are compiled by a P4 Compiler targeted for a particular data plane architecture. It models an abstract forwarding element which generalizes how packets are processed in different forwarding devices. This abstract forwarding model is targeted to the hardware on top which a common language (P4) is used to express how packets are processed.

At minimum P4 assumes the following switch abstraction. Packets are switched via a programmable parser which parses headers of the packets. This is followed by multiple "match – action" stages which are used to match and apply entries from action tables which are representation of the forwarding information base in the forwarding element. These stages can be in parallel, or series or any combination of those, the order is implementation specific. After processing from these stages the packets are reconstituted into to the output data link format by the deparser from which they exit the data plane.

The programmable parser is the core of the protocol independent nature of P4 itself. The headers are specified by the user P4 application and the match action stage abstraction determines how packets are modified by determining which tables the packet matches against to apply the required action, where the action is the resulting modification of a header field.

The idea behind these efforts is to provide flexibility to the application developer in defining how packets are processed regardless of the underlying hardware [15]. The forwarding plane hardware providers, on the other hand, may distinguish their devices on cost, performance and power metrics, while providing support for standard programming abstractions.

2.2. Forwarding Architectures

This section gives an overview of different types of forwarding architectures. Regardless of architecture design, a forwarding element as shown in figure 1 accepts a stream of packets. After basic header extraction and validation it looks up the route in the forwarding table. The lookup results in modifications to the packet headers and appropriate checksum updates. The resulting modified packet is output to the appropriate interface of the device. Forwarding elements may perform other packet processing functions depending upon the deployed application like traffic

shaping, QoS or payload inspection, this has given rise to different implementations like Soft switches, Network Processing Units and Pipelines.

2.2.1. Soft Switch

Soft Switches are widely implemented inside datacenter servers completely running in software within virtual machine hypervisors like XenServer [16], Vmware [17] or Hyper-V [18]. They are typically deployed for networking virtual machines by providing a virtual network interface which is mapped to the physical network interface of the host machine. This allows virtual machines to access the network in the same way as a physical machine would. Soft switches can also be used to form purely virtual networks if all of the virtual machines are completely within the datacenter subnet and don't require outside access.

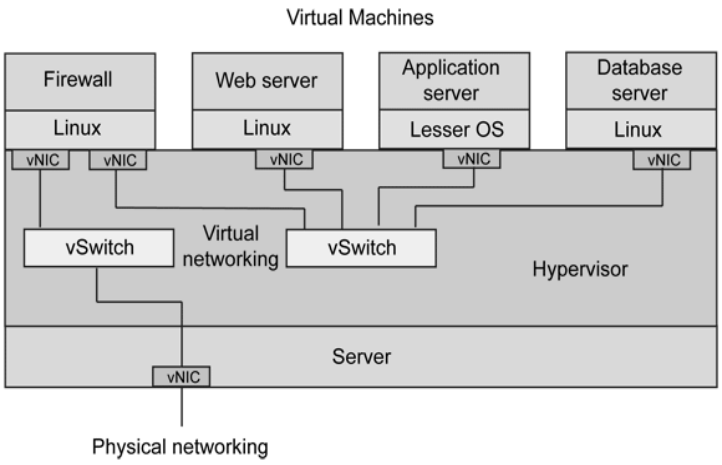

Figure 3 Virtualizing switches embedded in Hypervisors [41]

The fact that soft switches are virtualized themselves allows them to be distributed also across various physical machines which simplifies network management as there is no need to create identical switches for each physical machine in this scenario; however it also increases the computational overhead on the host machine and can only scale to certain size to have realizable network latency.

Open vSwitch (OVS) [19] is a popular open source virtual switch. Proprietary solutions include VMware vSwitch [17] as part of the Esx/Esxi hypervisor networking fabric and Hyper-V Virtual Switch from Microsoft.

2.2.2. Network Processor

Compared to soft switches network processing units are dedicated hardware geared towards particular networking applications. NPUs tend to strike a balance between the rigidity of full blown AISCs and the flexibility of soft switches. They are widely used in edge routers for performing high-touch functions like:

1. **Pattern Matching:** Accelerated regex functions to look for certain streams used in DPI.
2. **Forwarding and Routing Lookups:** Perform routing lookups and modify fields in packets using algorithms optimized for IP addresses and lookup data structures like tires or hash tables. To speed up certain types of lookups some NPUs use TCAM for variable length prefix lookups for L2-L4 processing or simple CAM for fixed-length MAC address lookups in L2 applications.
3. **Queue Management:** Quick allocation and re-circulation of packet buffers to implement QOS, ACL, Traffic policing & shaping.
4. **Encryption and Decryption:** Accelerated hardware provides cryptographic services for IPSEC by authenticating and encrypting packets of a session.
5. **L2/L3 Header Processing:** Header updates like port numbers and subsequent header processing that entails for checksum calculation and CRC updates to implement NAT.

NPUs are also used as line cards and are used to offload networking operations that are best done in software. Due to the inherent parallel nature of the workload, NPU architects use a large number of multi-threaded RISC cores to exploit data parallelism, while providing the flexibility of software programming. Figure 4 illustrates the high-level architecture of a simplified

NPU, inspired by the SNP 4000 architecture [16]. To form the rest of the packet processing pipeline NPUs consists of configurable hardware units/co-processors for parsing, scheduling, reordering, traffic management and deparsing of packets. It comprises of multiple identical processing clusters, each made up of a set of general purpose processing cores, memories and hardware accelerators.

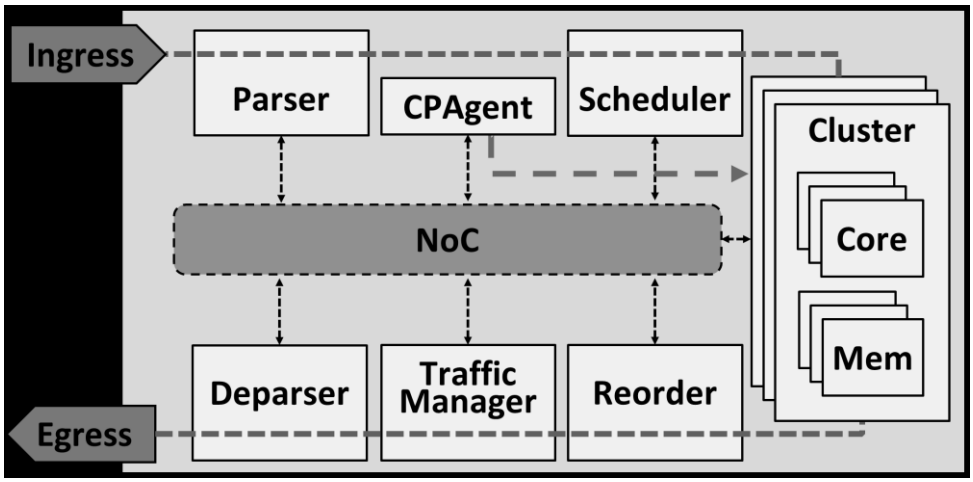

Figure 4 Typical Network Processor Architecture

2.2.2.1. NPU Packet Flow

The control plane populates the match table entries in the NPU's memory via an agent. This agent is typically part of the background processing cluster which by itself is not a part of the NPU's packet processing pipeline, but performs background tasks like communicating with the control plane for FIB population, handling match table misses, reporting network state and etc. NPU's receive packets at the ingress interface which are sent to the parser for header 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. All of the cores in the clusters share the same bank of physical memory local to the cluster.

Since the bulk of the processing takes place on the processing clusters and the requirement to communicate with the same set of modules from all of these clusters NPU's naturally lend themselves towards employing on-chip networks for communication between different elements to keep everything scalable.

The majority of the packet latency results from the search-and-lookup operations in memory from the processing taking place in the parallel cores in the cluster. 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.

2.2.3. Reconfigurable Pipelines

Pipeline architectures are currently theoretical and no current design exists in silicon one such is proposed in [5] It deploys the packet processing as a literal pipeline unlike the distributed architectures of NPUs. It employs parsers like the NPU which generates a packet header from packets arriving at ingress. The packet header travels through the pipeline in order on a wide header bus, with each stage executing a match action table. The match part of the stage uses TCAM or hash tables in SRAM for selecting the appropriate action in single cycle. The possible actions for a match-action table are preprogrammed as very wide instructions and stored in an SRAM section, dedicated to the relevant pipeline stage. A VLIW action processor executes the selected

instructions, and passes the packet header to the next stage. TCAM is used for wildcard matches and hash tables for exact matches.

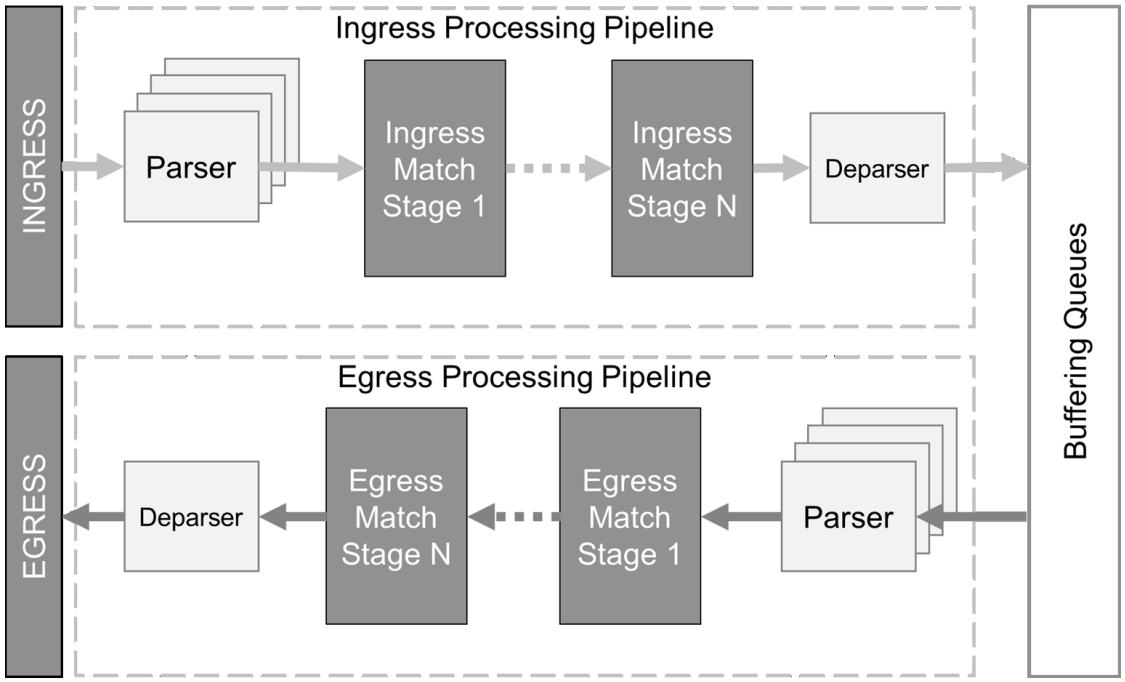

Figure 5 Reconfigurable Match Table Pipeline Architecture

This obviously puts the constraint that the application running on the VLIW action processing units should ultimately be able to be expressed as a set of these match action stage memory lookups, which means unlike NPUs RMT cannot be used to perform higher-level functions like encryption/decryption locally. In RMT, the packet latency is constant and depends on the number of stages.

2.3. Simulation Techniques

Simulation using virtual platforms has become a keystone for system-level design and validation by system architects. Architecture is replicated in software not only for just functional validation but also for early exploration into architecture design options and trade-offs, and most importantly to enable pre-silicon software development as time to market windows grow smaller with each development cycle. The techniques of software simulation used to be specific to application domains. Slow cycle accurate level modelling has been the purview of high performance computing domains focusing on complex architecture exploration, whereas the more resource constrained system on chip embedded domain privileged the co-design of hardware and software via fast and loosely timed transaction-level modelling. It is the recent convergence of these domains that demands for the confluence of design practices.

Function level modelling is just used for validation purposes to validate hardware against the initial algorithm proposal by comparing output traces under test for the set of same inputs. This however gives us no knowledge of the design of the system. Once algorithms are broken down to their structural components, we can model the interactions between these components as simple transactions as shown in figure 6. Coarse grain timing can be added to components for early performance and timing estimation. This allows architects to make fundamental decisions about the system like number of processors, algorithm characteristics, memory architecture which shape the eventual implementation of the target design. Going more detailed instruction and cycle accurate models simulate the model close to RTL, this allows the target software to run unmodified in this simulation. Although it is the most exhaustive simulation it is also the most expensive

simulation due to the level of detail and modelling effort required. It is of the used for debugging at the instruction level and verification of silicon synthesized from RTL.

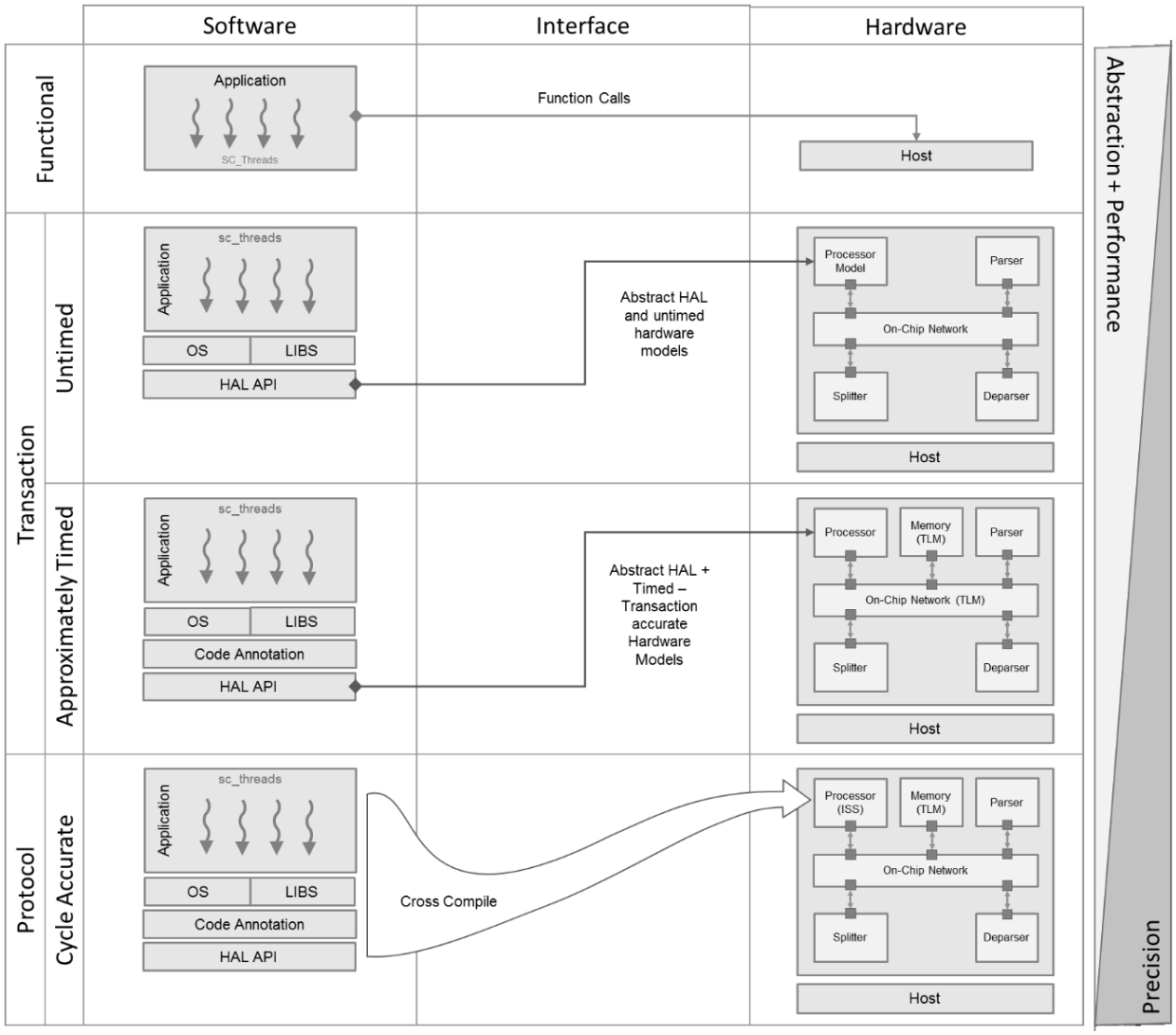

Figure 6 Comparison of modelling at various abstraction levels

Forwarding planes subsystems can be categorized into two types Fixed Function co-processors, which are specialized components that handle a single task and Processors which are used to perform general purpose processing on the packets.

The behaviour of fixed function components in forwarding planes like reorder controllers, deparser and traffic managers can be easily modeled as they are designed to maintain a fixed throughput for packets passing through them. In this section we will be focusing more on simulation techniques for modelling processors and applications as they from the majority of the sub-systems that contribute the most towards the variation of packet latency in any type of forwarding plane architecture.

2.3.1. Instruction Set Simulation

Instruction set simulators allow software execution of a *target* application program on a host machine by mimicking the execution behavior of the target. ISS are used to validate compiler and architecture design and evaluation during design space exploration.

ISS mimic the behaviour of the *target* at the instruction level to achieve this they mainly fall under three classes. The accuracy of such simulators broadly falls under instruction accurate and cycle accurate. Instruction accurate models usually target not simulation accuracy but rather application development; where cycle accurate simulators target simulation accuracy for architectural debugging and validation. Cycle accurate models are used for synthesis and are fairly close to RTL level models. However due to the excessive level of simulation detail at this modelling abstraction, the simulation speed is quite slow which combined the extensive modelling effort required hampers high level architectural exploration of SoC design and evaluation.

Since there are no *published* works for packet processing instruction set simulators this overview solely focus on work done in modelling of processors in the context of ISS simulation.

2.3.1.1. Interpretive ISS

In an interpretive simulation of a processor model simulation happens in a loop, instructions are fetched, decode and executed one by one sequentially as show in figure 7. The target instruction is translated to host instructions or equivalent host-compiled functions to be executed on the host platform.

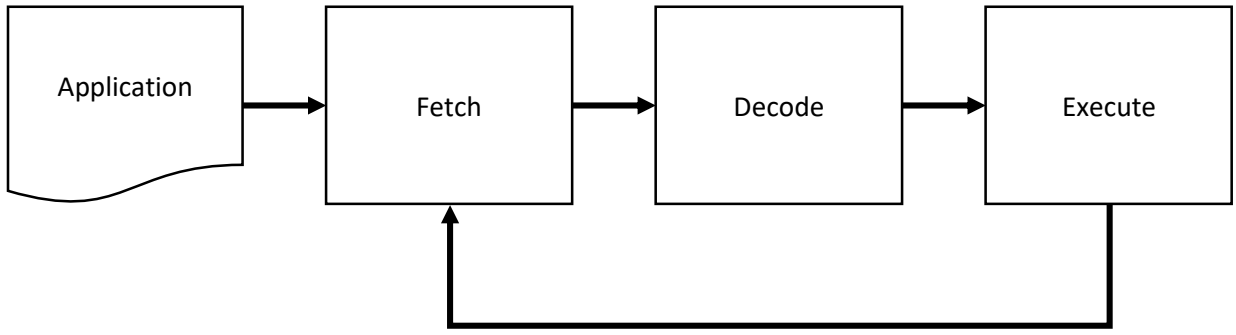

Figure 7 Interpretive ISS simulation flow

Interpretive ISS might be the most flexible approach but is also one of the slowest approaches due to it sequential nature of evaluation of each instruction. SimpleScalar [20] is such an example of an interpretive ISS, The *sim-outorder* tool is the lowest abstraction level in the toolkit which provides a cycle-accurate model of a processor. The simulator has seen wide adoption by researchers but it seems that both development and support have slowed down significantly, the last update was more than two years ago at the time of this writing.

Gem5 [11] is another simulation framework that provides processor models at different abstraction levels including various architectures of cycle-accurate processors as shown in Table 1.

Table 1 Comparison of full system cycle accurate simulators

Reference	Simulator	Accuracy	Supported processor architecture	Licence	Dev/Support
WindRiver [21]	Simics	Functionally-accurate	Alpha, ARM, MIPS, PowerPC, SPARC and x86	Private	Yes
Yourst [22]	PTLsim	Cycle-accurate	X86	Open	Yes
Austin et al. [20]	SimpleScalar	Cycle-accurate	Alpha, ARM, PowerPC and x86	Open	No
Imperas[12]	OVPSim	Instruction-accurate	Open Cores Open RISC, ARM, Synopsys ARC, MIPS, PowerPC, Xilinx MicroBlaze and others	Open and Private	Yes
Binkert et al. [23]	GEM5	Cycle-accurate	Alpha, ARM, x86, SPARC, PowerPC and MIPS	Open	Yes

Note: Reprinted with permission from Accuracy Evaluation of GEM5 Simulator System

Anastasiia Butko, Rafael Garibotti, Luciano Ost and Gilles Sassatelli

The major drawback of interpretive ISS is the extremely slow simulation speed, which results from intensive decode of target instruction at run-time; putting that in the context of a whole system platform, simulation is extensive but very time consuming at the cycle or instruction abstraction level.

2.3.1.2. Statically Compiled ISS

Statically compiled instruction set simulators decode and translate the *target binary* at compile time and extract tracing information for instructions. This eliminates the fetch and decode step compared to interpretive ISS. However this approach may work for simple systems it falls short to account for branching and mutable code. This technique requires application code to be

completely available at compile time, which is often not the case for languages supporting dynamic runtimes.

2.3.1.3. Dynamically Compiled ISS

Dynamically compiled ISS overcome the limitations of statically compiled ISS by dynamically translating or *retargeting* target instructions into host instructions. Instructions are translated in to simpler micro operations which are usually precompiled for the host platform which the target architecture is running on.

QEMU is such an emulator that achieves this via binary translation (or binary retranslation) [24]. This allows QEMU to run unmodified target operating systems. QEMU alone provides only virtualization and emulation features for CPUs, which allows functional simulation and application development, but it however does not consider timing and impact of architecture layout on software execution. It is designed for speed: only a valid x86 API behavior must be reached, which means not necessarily with the right number of cycles or even in the same pipeline order.

D. Thach et al. [25] introduces a technique for cycle count estimation in QEMU via timing analysis of the pipeline. It has two passes first it statically calculates pipeline timing prior to simulation and at run-time it uses that data to account for branch prediction/divergence. The authors report results for an ARM processor a 26% simulation error rate against an average of 10% to actual hardware implementation. They also report a slow down by a factor of 3.37 times compared against an equivalent QEMU functional simulation.

OVPSim [12] is another simulation tool offered by Imperas Software that includes a large number of functional model of processors like ARM, Xilinx Micro blaze, MIPS, Power PC. It allows

simulation of these multiprocessor platforms containing arbitrary local and shared-memory topologies.

OVPSim implements its virtual platforms which run cross compiled programs on a semi-host dynamic linked library which does the dynamic binary translation of the cross-compiled application binary translating target instructions into x86 instructions on the host machine. The processor models provided are instruction-accurate.

2.3.2. Host Compiled Simulation

Virtual platforms categorized as host compiled simulations are usually models at high levels of abstractions which allow them to provide fast evaluation but rely on coarse-grain estimation or worst-case static analysis.

Host compiled models typically start at the algorithmic level and are then modularly decomposed to the underlying hardware structure. [26] It is at this abstraction level that subsystems are defined and fleshed out to identify internal functions and the relations between them. Figure 8 shows the design process from high-levels abstraction to the gate level.

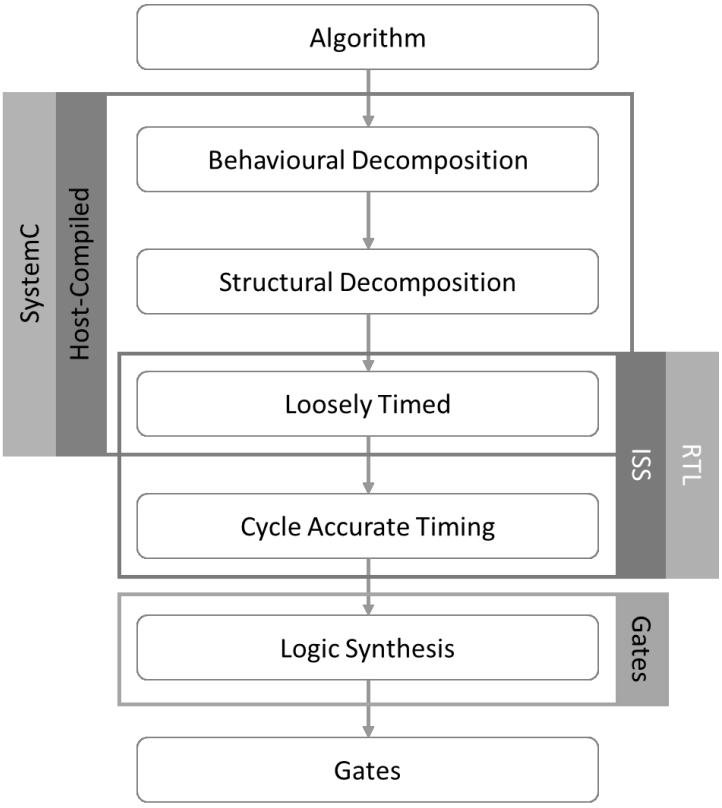

Figure 8 Design flow from algorithms to silicon

Fundamental decisions about the system are made like number of

processors, algorithm characteristics, memory architecture, data connections, software stack which shape the eventual implementation of the target design. Communication between subsystems can be represented by functions calls or at the transaction level between sub-systems, these transactions can be loosely timed for granular timing and performance exploration. Modelling at the transaction level allows models of different abstraction levels to co-exist in a single simualtion which allows more flexible IP resue and faster simulation by refinment of only “intersting” details.

Most models at these abstraction levels are for early design space exploration and application development with functional validation. At these levels processors are modelled as just modules with application threads which execute natively on the host and it is this native execution that gives host-compiled simulations the speedup over cycle-accurate simulations. Although good for functional validation they however do not directly allow performance estimation of the application on the target as only transactions between modules is modelled. To model the execution of host application code it is usually annotated. Tools like LLVM allow intermediate passes on compiled IR code to allow back annotation of statically determined low-level timing estimates. This still allows for fast simulation speed but improves the quality of the simulation as now application behaviour effects simulation timing. The limitation of Host-Compiled simulations is that unlike ISS they do not capture memory transactions as execution is directly on the host. This results in incorrect modelling semantics for applications which are heavily memory IO bound.

Given how integrated todays systems on chip are, they are no longer a simple dumb cluster of RISC processors or single beefy VLIW processors, functions are offloaded to a variety of optimized co-processors for faster and power efficient computation and all of these modules must work in tandem with each other to function. Full-blown simulation for a complete system at the cycle level would be too slow and in most cases is not possible because for most modules cycle

level simulators don't exist (think base-band processors, cryptographic AISC key hashers, etc.), since most of these modules start as high-level algorithms, which are eventually implemented in hardware. To quickly evaluate and make decisions host-compiled simulations allow full system simulation with speed and relative accuracy.

2.4. Summary

In this section we covered various implementations of forwarding planes from fully virtualized switches like soft-switches to high-speed low latency processing approaches for packet processing using distributed architectures like Network Processing Units or heavily pipelined architectures like RMT. Each platform makes its trade-offs between implementation complexity and application flexibility.

Next we took a look at simulation methodologies, instruction set simulation although the most detailed approach also has the most expensive modelling effort compared to Host compiled simulations which greatly inhibits the use of ISS as it does not offer the flexibility to evaluate different variations of models for comparative analysis needed to evaluate hardware design choices. Also due the extensive depth of simulation application development is unfeasible on ISS even with adaptive dynamic binary translation which sacrifices accuracy of simulation for speed.

In the next section we will be taking a look at how different forwarding elements can be modelled and aim to solve some of the challenges of host-compiled simulations.

Chapter 3: Modelling Forwarding Architectures

We use SystemC [10] and TLM [11] to model a host-compiled model of the hardware and the low-level software services of the forwarding device. The logic for the modules in the forwarding device is written in C++ by the user. The packet processing program to be executed by the programmable cores in the forwarding device is compiled from a P4 description and linked with the SystemC model of the forwarding device.

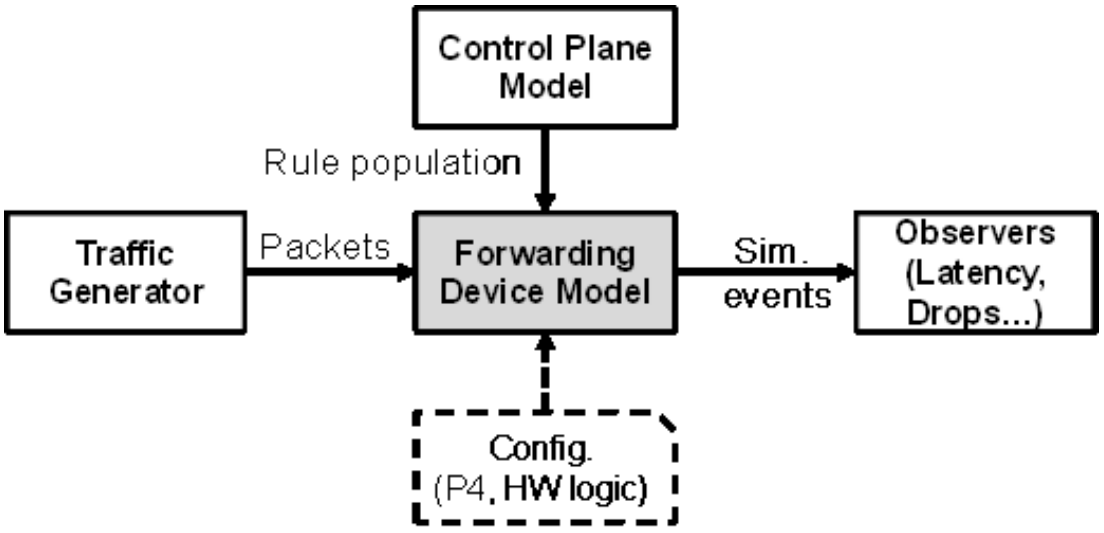

Figure 9 Simulation Environment

The control plane model populates the memories in the forwarding device model with match-table rule entries. The final executable SystemC model is stimulated by a traffic generator that feeds the model with a simulated stream of incoming packets. At run time, the model generates simulation events that are processed by user defined observers to derive metrics such as packet latency, drops, and energy consumption as shown in figure 9.

3.1. Transaction-level Modeling in SystemC

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. Transaction Level Modelling (TLM) is a modelling standard from OSCI [27]. TLM 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.

In RTL, simulation is synchronized based on a clock. In a TLM, this "synchronization" takes place when data is exchanged between two modules, which means a clock is no longer needed for simulation. All the processes, state changes, data movements, and calculations that occur in a particular model or between two units are known as transactions. Since this transaction represents the sum total of the processes that occurs in a system for it to take place it begins at a particular point in time and ends some time later. This bounds the time period in which the transaction takes place, thus introducing approximate timing to the system.

Transaction level models represent components as a set of concurrent, interactive processes that compute and characterize their behavior. The models thusly describe complex systems at a high level of abstraction. To separate communication from computation transactions are carried out over an abstract channel, the definitions of these channels is standardized in TLM [28] but the implementations are carried out by the designer, this allows interoperability and reuse of models as the interfaces remain the same for components modelled in TLM. For faster simulation speeds, we can move to higher level transactional level models which have minimal details. These models can be improved over time or swapped with detailed models for more accurate simulation.

3.1.1. TLM timing annotations

Since TLM transactions represent the sum of the activity in a model needed to carry out that transaction. TLM presents us with the following abstraction levels:

- Untimed

  This is the programmers view, components are modelled as processes and the transactions between components are represented using function calls. This allows functional validation but does not give any performance estimation of the architecture.

- Loosely timed

  This builds upon the untimed model, each transaction still completes in one function call. Timing is introduced by just two points, the start and the end. Transactions can also be temporally decoupled by letting process run ahead of simulation time in this case as shown in figure 10. The collective time is consumed at the end of all transactions which speeds simulation since it runs ahead of the master simulation clock. The standard also allows transactions to bypass the transaction-based block-to-block interface entirely and have direct access to areas of memory within a target function, again to accelerate simulation.

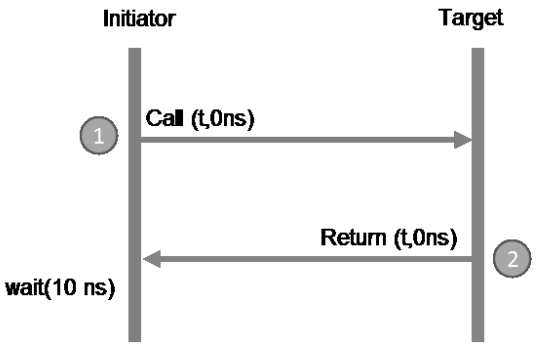

Figure 10 TLM - Loosely timed annotation – temporal decoupling.

- Approximately timed

AT attempts to achieve cycle-count-accuracy by annotating each step in the transaction as shown in figure 11. Each transaction is represented at the minimum by 4 timing points. More timing points can be added for more accuracy. However this means that processes must run in lock step with the master simulation clock time. It achieves relative accuracy of cycle-accurate models but does not incur the overhead cost of simulation all wire and pins of the model.

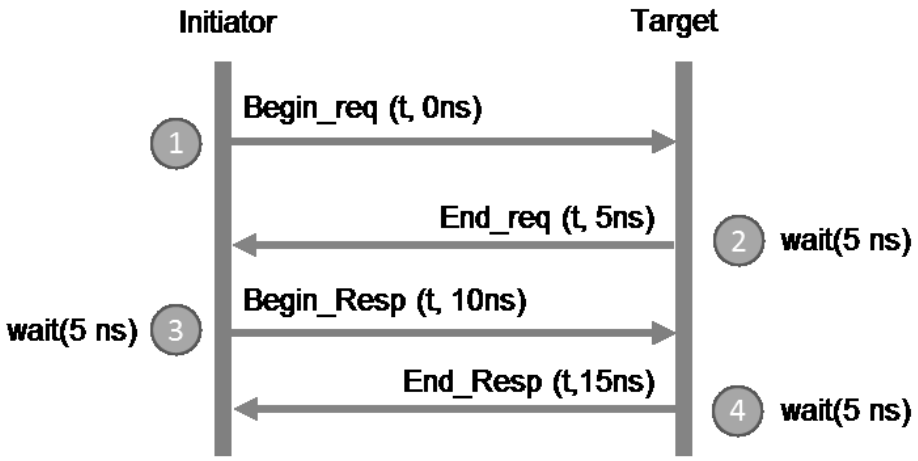

Figure 11 TLM Approximate timing annotation points

3.1.2. TLM Interfaces

TLM is all about interoperability of modules and get them talking to each other. In order to pass transactions between initiators and targets TLM provides the core interfaces for TLM sockets [28] as shown in figure 12. Initiators send transactions through initiator sockets, where targets receive them through target sockets. These sockets encapsulate communication between modules for both directions of communication. An initiator socket is implemented as a *sc_port* that is coupled with a *sc_export* on the side, whereas a target socket is a *sc_export* coupled with

a *sc_port* and we know a *sc_export* is actually encapsulating a *sc_port* with a *sc_channel* in it. The bind operator of the socket binds port-to-export and export-to port in a single function call. This allows the initiator and target to have bi-directional communication over the same socket and eliminates the need for implementing two separate channels for each direction. This convenience is a feature of sockets.

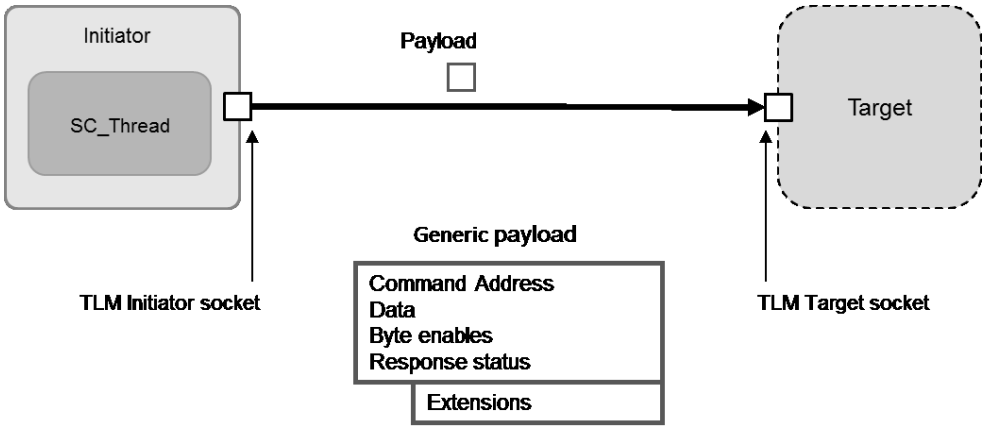

Figure 12 TLM Socket communication

The data sent over the subsequent link is carried in the TLM generic payload format, which already defines standard fields for data, addresses, commands, etc. These fields are information that is usually passed around in memory-mapped schemes as that TLM is geared toward providing interfaces for modelling busses and memory mapped systems.

Sockets already export a base protocol that handle the handshaking involved to send and acknowledge receipt of the payload by defining a set of points that mark the beginning and end of a request and a response. Just like the different timing notions these interfaces have different purposes.

3.1.2.1. Blocking transport interface (b_transport)

- Includes timing annotation.
- Typically used with loosely-timed coding style.
- Forward path only – Initiator is blocked until return form *b_transport.*
- Allows direct memory access within the target function by passing a *dmi* pointer.

3.1.2.2. Non-blocking transport interface (nb_transport)

- Includes timing annotation and transaction phases.
- Typically used with approximately-timed coding style
- Called on forward and backward paths

Figure 13 illustrates the SystemC TLM representation of a simple design with a single processor core (core0) connected to memory (mem0). The application layer (applayer) and hardware abstraction layer (hal) are instantiated as sub-modules inside core0.

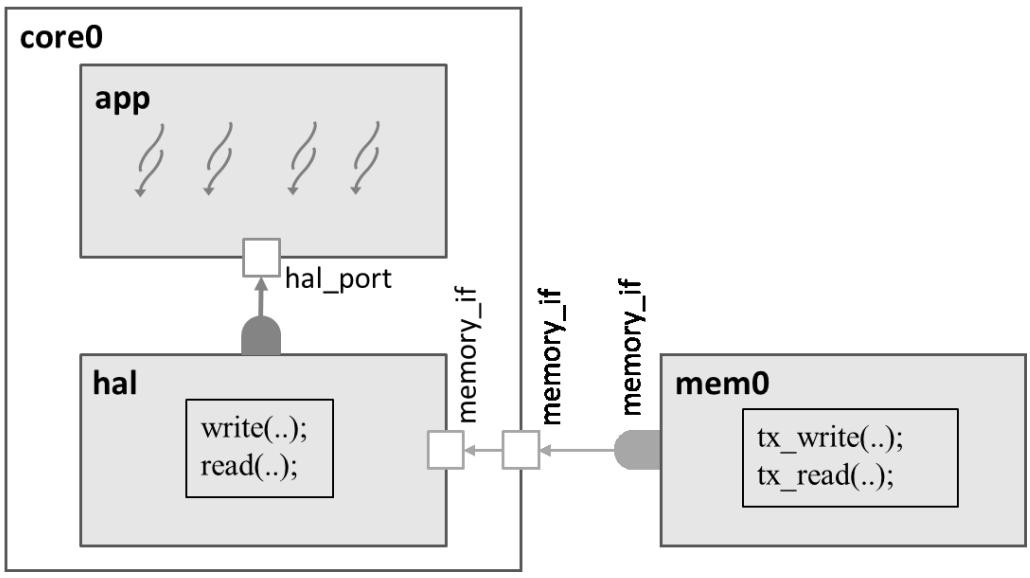

Figure 13 High level modelling of a processor.

Threads inside the application layer use a port (hal_port) to access the memory read and write services provided by the hal module. The hal, in turn, implements these services via the memory interface ports (memory_if) that connect it to the TLM memory module (mem0) that implements the hardware memory services and maintains the memory state. The threads in the applayer execute natively on the host, resulting in much faster simulation than an interpretive instruction-set simulator.

3.2. Modelling Memory

In our example above in section 3.2 we defined a communication element MEMORY which implements the memory_if interface, this interface defines functions for read and write as shown in listing 1.

Listing 1 Virtual functions declared by the memory interface

```
0: virtual void tx_read(tlm::tlm_generic_payload& trans)=0;
1: virtual void tx_write(tlm::tlm_generic_payload& trans)=0;
```

These functions accept a tlm transaction object which has fields for all of the information required for the transaction like virtual address, pointer to data on host memory and size of transaction. This Memory module (figure 14) provides the implementation for functions shown in listing 1.

For a module to read from an address in our memory module the initiator would simply call the *read* function of its *interface* which is connected to our CE and pass it a populated transaction object as shown in figure 14 with the *target* address and a *host pointer* to an object that the CE should populate the value from the *target* address.

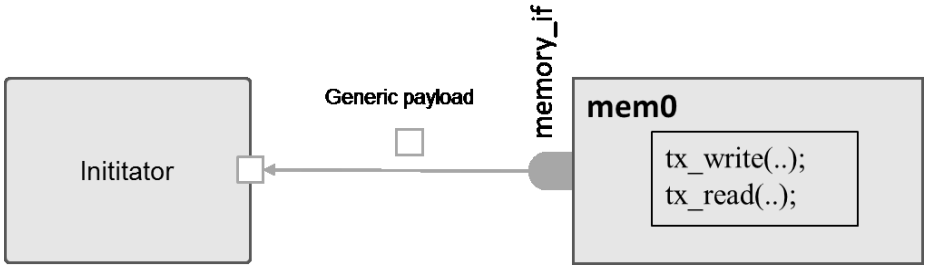

Figure 14 Memory CE interface

The function which is defined in the CE Memory would be called and it would execute its body shown in listing 2. The function accepts a TLM generic payload which has the memory address, a *host* pointer to return the data object, size of the transaction and command type. It performs address bounds checking to ensure a valid address and returns the value at the data pointer from its internal map of *target addresses* to *host pointers*. Similarly vice versa for the write operation except this time it also uses the transaction length to determine how many bytes to write from the address.

Listing 2 Memory CE implementation of the read function.

```
1: virtual void read(tlm::tlm_generic_payload& trans) {
2:
3:    // check address range
4:    if (!ValidateAddr(trans.get_address();)) {
5:      RaiseError("Given transaction out of range:");
6:    }
7:
8:    if (trans.get_command() == tlm::TLM_READ_COMMAND ) {
9:          trans.get_data_ptr() = mem.at(trans.get_address());
10:         // response status to indicate successful completion
11:         trans.set_response_status(tlm::TLM_OK_RESPONSE);
12:   } else {
13:         trans.set_response_status(tlm::TLM_ERROR_RESPONSE);
14:   }
15:  }
```

Upon complete execution of the function control would be returned to the Initiator where we can consume the time taken for this transaction (temporal decoupling as discussed in section 3.1.1). The *host data pointer* in the transaction object now points to the object at the target address we read from. Since this module is a communication element we parameterize properties of the module like memory size, read and write latency to a configuration file. This parameterization allows us to model various types of memories like SRAM, edRAM, DRAM each having its own characteristics of read, write latency and maximum capacities.

3.3. Modelling the Memory Controller

Traditionally memories are connected using shared busses or cross bars to processors but in distributed architectures like NPUs they employ networks on chip due to the fact that all of the distributed processing clusters have to talk to the same fixed-function modules. A memory by itself only understands addresses, conventionally these addresses are accessed by processors via the bus through an arbitration mechanism. Shared busses don't scale for processing cores inside clusters all of which need variable non-sequential access to the memories.

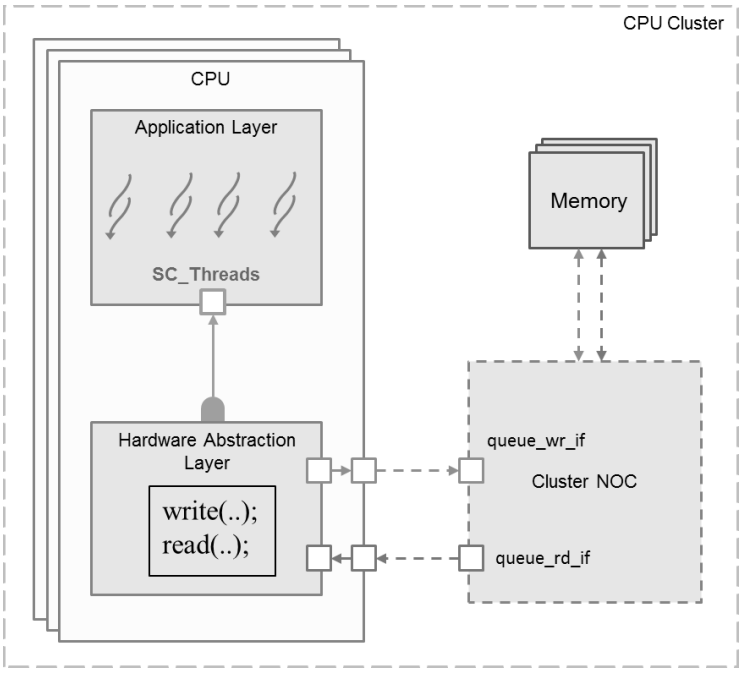

Figure 15 On-Chip network access to Memory

Also since each cluster has the same copy of the lookup data structures in their memories a distributed method for building these data structures in each cluster is needed. This is why the memories need to be connected to the on-chip network employed in the NPU. However the memory module cannot by itself directly connect with the on-chip network as it does not have the necessary interface.

Memory Controllers are the bridges between the two different set of interfaces. They are modeled as Processing Elements as they are active components that service requests from the on chip network interfaces *queueRdI* and *queueWrI*. The memory controller carries out the transactions to the memory over its m*emory_if* interface by converting the received request to *tlm_generic_payload* objects which the CE memory module can understand.

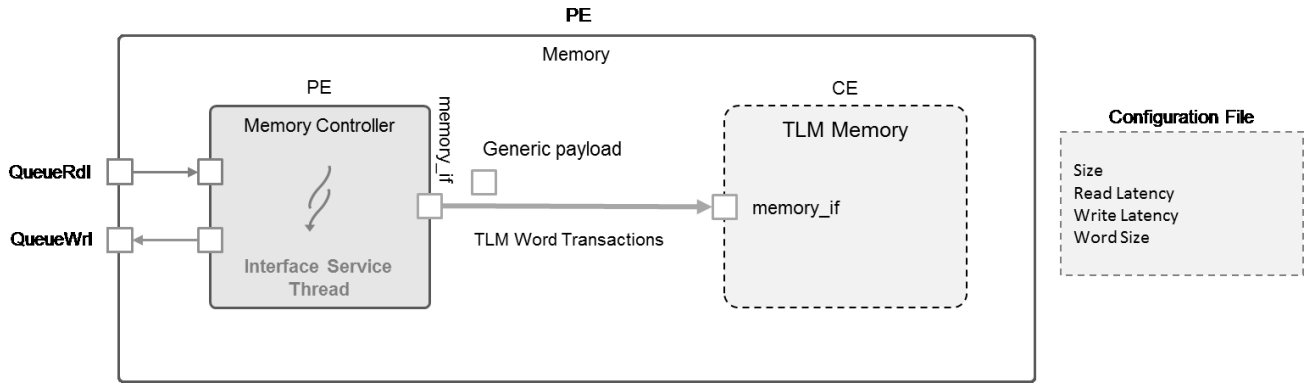

Figure 16 Memory PE

Memory Controllers handle requests from modules for retrieving or storing packet payloads at an address or just general read/writes for lookup tables stored in memory. To allow for faster simulation we temporally decouple memory transaction to the memory CE instance *tlm_memory* by consuming time after the transaction call has finished in the PE Memory Controller. In this case the memory module itself becomes a hierarchal PE for wrapping the whole module. This memory PE forms the basic block for the memory architecture of the NPU model.

3.4. Modelling Fixed Function Modules

Modules like Parsers, Routers and Schedulers are fixed function modules as they are solely implemented as dedicated hardware. The logic of these different modules is described simply in C++ which is executed in a systemc thread. The design pattern for describing the behaviour of modules is to have separate reader and writer threads as shown in figure 17 with internal buffers for storing/servicing transactions.

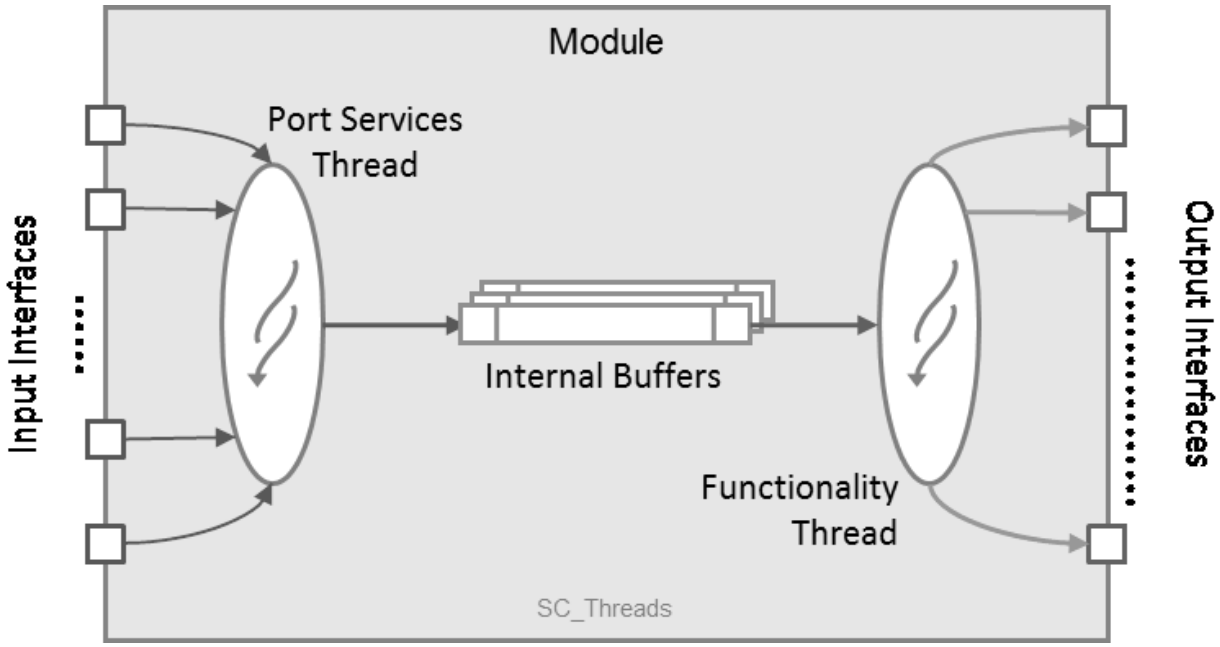

Figure 17 Design Pattern for fixed function modules

Modelling concurrency and concurrent behaviour can be tricky and often not so straight forward. Module behaviour modelled as single threads can often lead to modeling incorrect behaviour e.g. they can miss transactions on their interfaces which they may not in real life if they are stuck processing or waiting on another resource and such behaviour may only appear under certain test conditions which would make the model less robust in terms of modelling accuracy.

To solve this a systemc thread consumes these transactions from the input interfaces and places them in internal buffers.

Whereas another separate systemc threads models the actual behaviour of what the module performs on these transactions *(functionality of the module).* The port servicer thread initiates a systemc event notification every time it places a transaction in the buffer. The Functionality thread waits on this event notification when idle to resume processing by picking data from the internal buffer.

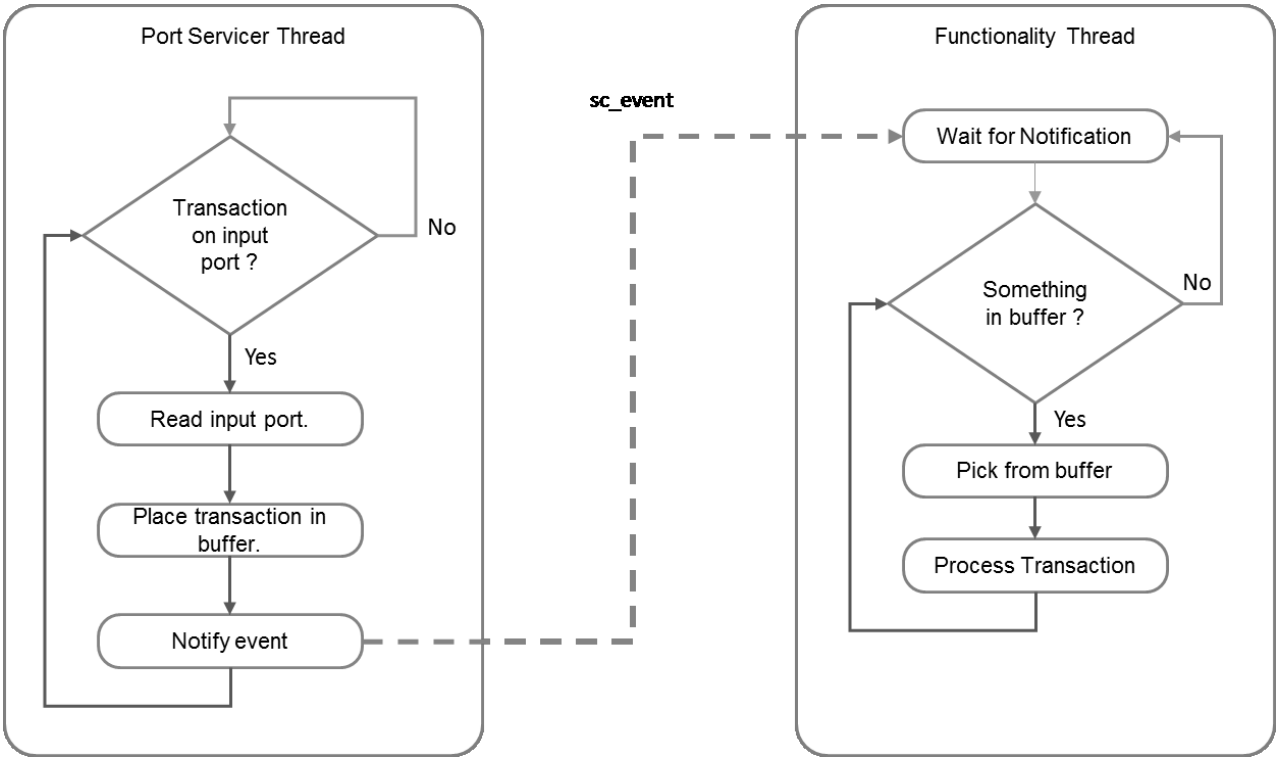

Figure 18 Behavioural design of fixed function modules

This avoids the design pitfall of putting all functionality in a single thread, as we now control the rate of servicing input ports in the port servicer thread by adjusting the size of the internal buffer. If the buffer is full then the port servicer thread can just drop the transaction or simply let the transactions accumulate at the link/queue (if the link models blocking to model

congestion) that the interface is connected to. The two thread design is the simplest example and scales very nicely in terms of having multiple processing threads.

3.5. Soft switch Model Semantics

The soft switch is a virtualized switch as discussed in chapter 2, and has the simplest modelling semantic. The softswtich is modelled as a simple module with the application executing inside the module in a single SC_Thread. The module has two ports one for input and output which the soft switch uses to process packets.

The soft switch model has no timing semantics and executes natively on the host in the SC_Thread, its traces are used to determine functional validation of other architectural models.

3.6. NPU Model Semantics

Our NPU model design deploys a distributed packet processing pipeline with a stream of input packets being read by the splitter module, which splits the headers from the payload and generates a packet descriptor representing the unparsed headers, this is sent to the parser over the OCN. The payload is dispatched to the requisite memory allocated by the memory manager. Parser parses the packet headers and populates the fields in the packet descriptor. Retargeted code from the P4 behavioral model is used by the parser to fill out the packet descriptor. It returns the number of states it took to parse the headers which is used to model single cycle latency for each parsed header. The parser dispatches the packet descriptor to the Global scheduler. The Scheduler deploys a FIFO pull model. Processing clusters request jobs from the Global scheduler which it services by forwarding packet descriptors to the cluster for processing.

The current design employs 8 processing clusters. Each processing cluster is comprised of 4 memory modules which represent the on-chip memory of the system and 8 processing cores which are 4 way threaded. The processing cores are managed by a cluster scheduler, which pulls processing requests from the global scheduler and assigns them in a round robin fashion to the processing clusters. The application executes inside the Processors as shown in figure 15 in SC_Threads. Data structures for longest and exact match lookups are populated for each cluster on the chip memory and spill over into a singleton memory module which represents the off chip memory in an NPU.

Due to the concurrent nature of processing any two packets being processed in the cluster on different cores, may encounter different delays due to lookups from the FIB in the memory and may leave the processing clusters in a different order than the one they arrived in. The order of packets is important for certain applications like in TCP under worst-case scenarios it would cause incessant retransmissions from the end nodes. Therefore, a reorder module is used to restore the ordering of packets. The traffic manager shapes packet flows according to their priority. Traffic managers are typically implemented as fixed-function units and consists of large banks of queues for quick-allocation and re-circulation. Deparser reconstitutes the packet with its processed headers and payload and sends it to the egress port where a SERDES module encapsulates it into to the egress data-link format.

3.7. RMT Model Semantics

The RMT model is based on the forwarding architecture proposed in [5]. It is composed of three top level sub modules: the ingress and egress pipelines, and the queues in between them as

illustrated in figure 5. The ingress and egress pipelines are both instances of the same module, which contains several sub modules:

- **Parsers**: Sixteen parallel parsers service the ingress port of the pipeline. The parser uses C code retargeted from the P4 behavioral model hence is programmable. The P4 code parses the incoming packets into a *packet header vector* (PHV) which is the internal representation of the parsed headers from a packet. It return the numbers the states it took to parse the packet. These number of states are used to model single clock cycle computation delay of a TCAM based state machine parser proposed in the RMT architecture [5]. All of the parsers feed processed PHVs to the Match Action stage pipeline.
- **Match-Action stages:** These represent the 32 logical match action stages each consists of 3 sub modules. The *selector* constructs keys from the PHV fields which are used by the second module the *match tables'* stage. The match stage depending on the key type performs lpm in a TCAM model or an exact match using hash tables. The lookup results in an action. This *action* is applied to the PHV fields in the third stage, which modifies the fields accordingly. The PHV is then forwarded onto the next stage in the pipeline. A PHV passes through all of the stages but is only processed by only those stages for which it is tagged this is done to implement the table-flow graph specified in the P4 program, we add a next_table field to the PHV. When a stage receives a PHV, it checks if its index matches the next_table index of the PHV. If there is no match, the stage simply passes the PHV to the next stage in the pipeline without modifying it. Since the RMT architecture executes a match-action stage in a fixed number of cycles, we model that by giving each stages a fixed processing delay.

- **Deparser:** This is the last module and consumes all of the PHVs processed by the pipeline. Like the parser it uses the deparsing function retargeted from the P4 behavioral model to piece back the processed PHV with its payload into a packet.
- **Queues:** These are buffering queues that store processed packet from the ingress pipeline to be consumed by the egress pipeline.

Although we model the ingress and egress pipelines as two separate modules. In reality the RMT architecture implements these as logical pipelines with logical match action stages for each pipeline on a single hardware pipeline with a fixed number of actual match action stages in which the logical match action stages are interleaved with each other to form the ingress and egress pipelines.

3.8. Summary

In this section we presented modelling methodologies of SystemC and TLM and how we leverage them to model different hardware modules that a forwarding element might consist of like processors, memories, controllers, and fixed-function elements e.g. splitter, parsers, on-chip networks and Traffic Managers. We build upon this by describing the modelling semantics of various forwarding architectures like Soft switches, NPUs and RMT to create fast, executable simulation models for power, performance and pre-silicon evaluation. Although we have modelled memories the most evident modelling violation is that since it is a host-compiled simulation all memory transactions are directly happening on the host and not to our hardware memory models. In the next section we propose a solution to redirect memory access to our memory models.

Chapter 4: Hardware Abstraction Layer Modelling

The C/C++ code targeted for the cores is either provided by the user or generated from P4 compiler from the P4 description. The application code is wrapped inside a SystemC thread (SC_THREAD), spawned inside the application layer (SC_MODULE) corresponding to the core. The timing model for this code may also be provided by the user by adding SystemC wait statements at the source level. This is both cumbersome and impractical.

The most evident modeling abstraction violation is that, it hides the memory transactions from the core as memory transactions are happing natively on the host as shown in figure 19 and not through our memory models.

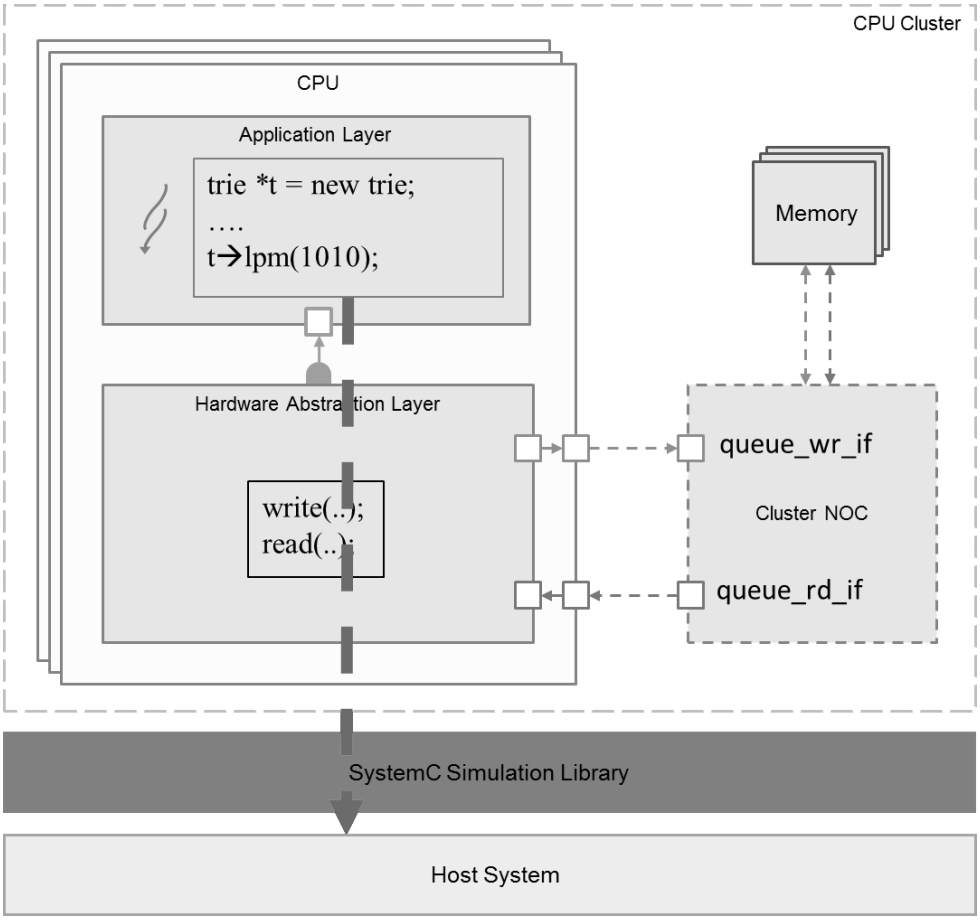

Figure 19 Application exclusively uses the host memory

If a variable is defined inside a thread, it is allocated and initialized on the host's memory directly because SC_THREADs execute as native threads on the host operating system. Any variables that are allocated are in the context of the host OS. Such abstraction may be acceptable for thread local variables that are used for relatively inexpensive action processing. However, this abstraction is insufficient for trie accesses which are made during search and lookup operations for implementing the match operations as these lookups in memory are the major contributors to overall packet latency.

Therefore we require a memory indirection mechanism for our lookups that redirect these memory access of a C++ object to transactions to our Memory models during simulation without requiring the application developer to know about the underlying hardware architecture.

We achieve this by defining clear semantics for our memory hardware model:

- Hardware abstraction layer model instantiated in the CPU cores to carry out transactions to the memory module.
- A variable base class - TLMVAR, for memory redirection that use the hardware abstraction layer to initiate memory transactions.

The separation of the variable base class and HAL is an important distinction as it makes our indirection layer more flexible and less independent on the underlying architecture of the model, which allows it to be portable as it just has to call standard HAL functions which is up to the HAL designer to implement.

The HAL PE implements the memory allocation, read and write services for the application threads running on the CPU core. The objects using the base class use these HAL functions to

redirect memory access to the memory model. Listing 3 shows the HAL code for the write method which takes the virtual address of the object being written, a *host* pointer to the value and the size as inputs. It decodes the *target* virtual address to a physical address and the memory instance it needs to send the request to. Since HAL is directly connected the core's ports it initiates a transaction to the memory module via the on chip network.

Listing 3 HAL implementation of the write function

```
1:      void HAL::write (long vaddr, void*VAL, long size) {
2:            auto mem_id = lookup_mem(vaddr);
3:            auto mem_addr = lookup_addr(vaddr);
4:            auto p = lookup_port(mem_id);
5:            p->tlm_write(mem_addr, VAL, size);
6:      } // end hal write
```

This transaction is posted to the memory PE routed via the On-Chip Routers where it is serviced by the memory controller of that memory PE. The Memory controller as described in the sections above carried out a tlm_write transaction to the tlm_memory CE for the given memory address and then posts the reply to the transaction which is serviced by HAL and returns control to the application.

The HAL layer exports three service functions for use by TLMVAR:

- **Allocate:** This function takes the number of bytes as input and sends the allocation request to the memory manager which keeps track of free memory blocks in the address space.
- **Read:** This service function implements the read transaction that the CPU model has to make to the memory model to access the value at a particular address.

- **Write:** This service function implements the write transaction that is made to the memory model in the event of a write event. It takes the address, the data pointer and the number of bytes to write.

The HAL Allocate, Write and Read functions can be called directly from the application code itself from within the Application Layer PE. Doing this manually for every read, write is cumbersome. To preserve normal programming syntax and paradigms from the developer's perspective we implement a redirection layer for variables that does the job of calling HAL functions for us to route transaction to the memory module, instead of directly using the host memory.

4.1. Memory Access Redirection – TLMVAR

We define a variable base class, named TLMVAR, whose sole purpose is to redirect memory access using the HAL functions. All objects in the application that need to be explicitly modeled as residing in the hardware memory models are instantiated from types based from TLMVAR. To understand how to achieve this we need to take a look at objects in C++ and the C++ Memory Model.

4.1.1. C++ objects and the Memory Model

The C++ specification does not make reference to any particular compiler, operating system, or CPU. It makes reference to an *abstract machine* that is a generalization of actual systems [29]. It is the job of the compiler to concretize this abstract machine [30]. In C++ the smallest addressable unit of memory is a byte, and the memory available to a C++ program is one or more contiguous sequences of *bytes*. Each byte in memory has a unique *address*. This is all defined by

the C++ standard [31]. The C++ memory model is meant to be transparent to the user. For an object to exist it must be allocated and initialized first before it can be accessed and modified.

4.1.1.1. Allocation and initialization

We can divide the C++ objects in two categories based on how they are instantiated and allocated in memory.

a. C++ Classes

A class defines a user based data type much like a struct does. Whenever an object of a C++ class is declared the memory the *new expression* is invoked [32] which tries to allocate memory for the object either on the stack or on the heap by the use of the *new operator* upon successful allocation it calls the constructor of the class which initializes the data members of the class, data members of a class are laid out in memory in the order they are declared in [33].

Upon completion of the scope of the object the destructor of the class is class and the *delete operator* which frees all of the memory allocated. This is done automatically by the compiler and is known as *Resource Acquisition Is Initialization (*RAII) [34].

b. Plain Old Data types

Plain old data types (POD) refer to types such as int, char, double, float which come from C [35]. PODs don't have constructors or destructors and allocation operators like *new* and *delete* are not allowed by the standard to be overload for these types [36]. The compiler manages everything for these types.

4.1.1.2. Access operators

C++ objects are accessed via assignment and access operators such as subscript `a[b]`, indirection `*a`, address of `&a`, member of a pointer `a->b`, etc. These operators for C++ classes can simply be overloaded to redirect access to our memory model. However again for POD types the standard forbids the overload of operators [36].

4.1.2. Redirection for C++ classes

From the discussion above we can summarise that while C++ classes memory model is exposed to the programmer where can have redirection points by overloading specifically the allocation (*new & delete*) operators and access operators (*assignment, pointer indirection, etc.*).

C++ classes inherit from the base class TLMVAR which has these overloads. Since the order of construction is guaranteed by the C++ standard, TLMVAR new operator is invoked first followed by its constructor, which calls the underlying HAL allocate function with the *sizeof* operator [37] to get the number of bytes for allocation. Once this function call returns it stores the internal virtual address and returns control back to the constructor of the class of that object which initializes the data members of its class.

Similarly for access operators all operations call the underlying HAL read or write function depending the type of operation being performed is determined by the type of operator. From example the statement `A = B` for two objects A and B of *class a* and *class b* that utilize TLMVAR will invoke the access operator of *B* which causes a read operation first from the *target* virtual address of B and then subsequently a write operation to the *target* virtual address of A. TLMVAR overloads the following operators.

Table 2 TLMVAR operator overloads

Available Operator Overloads					
assignment	increment decrement	arithmetic	logical	comparison	**member access**

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a->*b`

Member access operators like member of object `(a.b)` and pointer to member of object `(a.*b)` are not available as they cannot be overloaded as defined by the C++ Standard. Majority of the overloads are formed by chaining the overloads for the member access (Table 2 column 6) and the assignment operator `(a=b)`.

4.1.3. Redirection for POD Types

Plain old types all follow the same allocation and access scheme as C++ classes do but the operators cannot be overloaded for these types as defined by the C++ standard [36]. POD types like *int, bool, char, double, float*, etc. form the basic blocks of forming user data types, and however we can encapsulate these data types by declaring a data member of that type within a C++ class which would allow us to implement redirect the memory mechanism. Declaring a new single class every time we want to use a POD type is very cumbersome. That is why TLMVAR is also implemented as a templated version, simply passing the POD type as the template parameter encapsulates the object and allows redirection while retaining normal access paradigms. A partial

template specialization for POD pointer types is also implemented that achieves the same functionality except upon dereferencing pointers it returns values stored at target memory addresses. Listing 7 shows how a POD variable can be modelled as an object in the hardware memory model.

Listing 4 TLMVAR support for POD types

```
0:    // Integer on host memory.
1:    int Obj_on_host_mem = 6;
2:    // Integer variable in tlm memory with a value of 5
3:    tlm_var <int> obj1_on_tlm_model(5);
4:    // Normal C++11 initialization paradigms hold
5:    tlm_var <double> obj2_on_tlm_model {7.5};
6:    // Update the value stored in tlm memory to six.
7:    obj_on_tlm_model = 6;
8:    // Combine memory fetch and writes with operator chaining. Value=17
9:    obj_on_tlm_model = obj2_on_tlm_model + 10;
10:   // Update a host value from a target memory location.
11:   Obj_on_host_mem = obj2_on_tlm_model.to_host<int>();
```

4.1.4. Assembling the SW stack

Data structures that are using TLMVAR are often nested and encapsulated object instances of those classes within the application C++ code that is executing in a SC_THREAD in the application layer. For TLMVAR to access the HAL functions to perform read/write or allocation it needs a pointer to the application layer module. SystemC provides runtime functions for dynamic threads to determine where they are within a module hierarchy by returning the current thread handle using *sc_get_current_process ()*. This returns the current process handle which is used to get a handle on the parent module this thread is being executed in.

TLMVAR inspects the ports from the module handle to find the hal port which it uses to call the appropriate HAL function. The HAL function then makes the corresponding transaction to the memory, which is defined by the HAL that the port/interface of the current module is bound to. The complete flow is represented graphically in figure 20. Listing 5 shows the write function of TLMVAR. Any allocate, read or write operation on a TLMVAR-derived object results in calls to HAL functions, which in turn is responsible for the appropriate memory transactions in the hardware model.

Listing 5 TLMVAR write implementation

```
1:   void TLMVAR::write (void *VAL, long size) {
2:          auto a = get_applayer(sc_get_current_process());
3:          auto hal_port = get_hal_port(a);
4:          hal_port.write(vaddr, VAL, size);
5:    } // end write
```

In order to further insulate the application developer from the hardware model details, we have developed a trie library on top of TLMVAR. The search and lookup in the application code use this trie library without having to explicitly create TLMVAR objects. Thus, with clear modeling semantics, we enable the efficient development of applications on the simulation model.

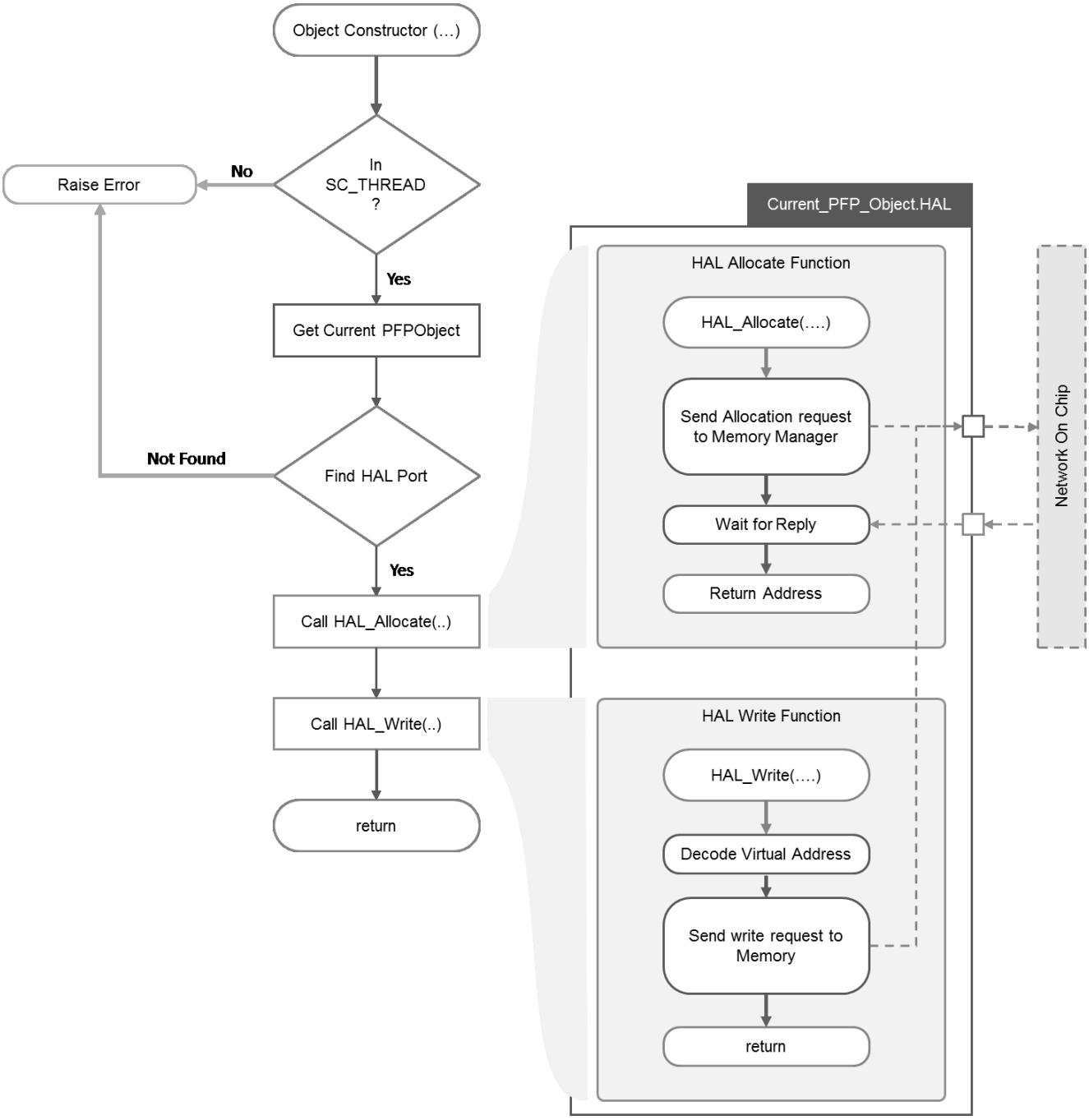

Figure 20 TLMVAR operation flowchart

4.1.5. Limitations of TLMVAR

TLMVAR is a compile time solution that exposes the C++ memory model, it however cannot support complex polymorphic data structures which have pointers to the same virtual table in its inheritance hierarchy i.e. in multiple inheritance cases when base classes share the same parent class in between themselves, this forms the dreaded diamond problem [38]. This is solved by virtual inheritance from those base classes. The compiler implements virtual inheritance by making the vtable pointer point to the shared virtual table; this allows it to go up and down the inheritance chain for any class[39]. Since the virtual tables are not exposed by any compile or run time expressions because the C++ specification does not specify virtual tables it is a C++ implementation detail by compilers. This means we cannot ascertain if in the inheritance chain of the derived class has there are multiple vtable pointers pointing to the same virtual table this again would lead to incorrect size and layout deduction. Moreover systems that do not have alignment requirements or don't pad structures and data structures that override the compiler's reordering of data members to pack data more efficiently will lead to incorrect size deduction for the data structure due to absence of reflection from C++ we cannot inspect the data members and what order they are laid out in the memory at runtime.

Some of these limitations are of the language itself but some can however be overcome by exploring to implementing a lower pass over compiled IR code using LLVM tools which annotate memory access originating from basic blocks by demangling names and unpacking data members of classes.

4.2. Integration with packet processing programs (P4/C++)

P4 provides a publically available compiler and a behavioural model for developing P4 applications. The P4 behavioural model is a runtime environment for P4 applications. A P4 application is compiled to a JSON representation which is parsed by this behavioural model to emulate the behavior of the application. This is mainly targeted as a soft-switch. We break up the behavioural model and compile it 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, which decouple the behavioural model runtime from its compiler functionality. These APIs, specifically, implement P4 initialization, parsing, table construction and lookups, P4 application execution and deparsing functions. These API methods are called directly from the user logic executing inside SC_THREADS from their respective PEs.

This allows the user to simply describe their switch implementation in the P4 language, which is then compiled by the *P4 compiler* to a json file that the *retargeted* behavioural model accepts. The P4 behavioural model takes the json file as input and uses it to parse and process packets according to the switch description. Furthermore, we re-implemented the search and look-up data structures used by the P4 application for longest prefix and exact matches on top of TLMVAR which further insulates the P4 application developer and abstracts away the memory redirection layer and architecture implementation. This allows the application model to accurately

capture the memory transactions of that *target* architecture during run-time all the while maintaining the same programming paradigms and abstractions from a developer's point of view.

4.3. Summary

In this section we presented a novel compile time solution for redirecting memory access to hardware memory models. This enables host-compiled simulations to leverage built in C++ language semantics to capture and redirect allocation and access of not only C++ data structures but also built-in C primitive data types with minimal run-time overhead while retaining normal programming paradigms and syntax from the programmers point of view.

In the sections covered by now we have looked at how to model hardware and software using SystemC and TLM methodologies, which is definitely a step up from modelling at the ISS abstraction level, but is still cumbersome as it requires significant re-working of the model for design exploration and evaluation.

In the next section we propose a methodology which leverages the ease and modular design of SystemC and TLM for automatic model generation from a specification, to reduce the modelling effort on part of the designers and developers.

Chapter 5: Automatic Model Generation

5.1. Forwarding Architecture Description

Although SystemC provides all the constructs needed to create an executable model of a forwarding device, it is still too detailed as a specification language for forwarding architectures. When exploring architecture design spaces SystemC models require extensive changes that have to be done manually by the user.

This limits choices into what models can be explored easily with the least amount of effort. To counter this we develop on the TLM and SystemC methodology discussed in the previous sections and put forward an abstract forwarding architecture description (FAD) language for the designer to describe the hierarchical structural model of the forwarding device. FAD provides a few simple declarative constructs to succinctly specify the hardware-software platform without having to create a detailed model in SystemC. Listing 6 illustrates the FAD for the design example in Figure 13. We use this example to explain the constructs in FAD.

Listing 6 FAD Example

```
1:    interface MEMI;
2:    CE MEMORY implements MEMI;
3:    service  HALS;
4:    PE HAL implements HALS {
5:          MEMI memory_if;
6:    }; // end HAL
7:    PE APPLAYER {
8:          HALS  hal_port;
9:    }; // end APPLAYER
```

```
10:    PE CORE {
11:         MEMI memory_if;
12:         HAL hal;
13:         APPLAYER applayer;
14:         bind applayer.hal_port, hal;
15:         bind hal.memory_if, memory_if;
16:   }; // end CORE
17:   PE TOP {
18:         CORE core0;
19:         MEMORY mem0;
20:         bind core0.memory_if, memory_if;
21:   }; // end TOP
```

5.1.1. Interface

The ***interface*** keyword represents the type of a hardware port for carrying out transactions. Interface types are purely abstract. The behavioral description of an interface is a set of pure virtual functions in C++. From the example above, our interface *MemI* (listing 6, line 1) defines the memory interface and the *transactions* that can be made to memory are either read or write operations. As shown in figure 21, our interface defines two functions tx_write and tx_read.

```
tx_write(..);
tx_read(..);
```

Figure 21 FAD Interface

5.1.2. Communication Element

Interfaces only describe what the transaction looks like, they do not define how the transaction takes place. Since interfaces are purely abstract port types; Communication Elements (CEs) are the concrete modules that provide those port types. The ***CE*** keyword is used to define the communication element type and the ***implements*** keyword defines the interface that the communication element provides (listing 6 line2). CEs are very simple elements which model passive memories or links between modules. CEs implement and define the implementation of the methods declared in interfaces. The CE contains the declarations of the functions *tx_write* and *tx_read*. This allows us to model *how* a transaction takes place. Keeping in align with the TLM methodology CEs separate the communication of a transaction from the processing of a transaction. A communication element is always the target of initiated transactions.

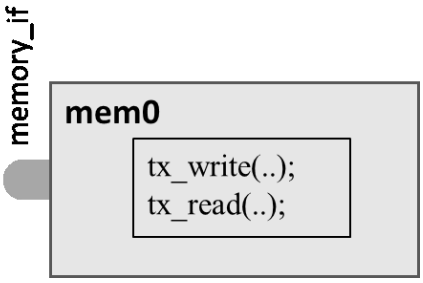

Figure 22 FAD Communication Element

5.1.3. Processing Element

The keyword **PE** defines a processing element type. It can represent either a hardware module, such as a parser or CPU cores, or software modules like application layers or drivers. PEs may be instantiated within another PE to express structural hierarchy (listing 6 lines 10, 11, 15). PEs are the initiators of transactions to CEs (listing 1 line 20). Interfaces instantiated within a PE (listing 6 lines 5, 12, 12) represents the ports of the hardware module and services represent API's that a software layer exports.

Processing elements are active components unlike CE's. Application threads can be instantiated inside PE's that model the behaviour of the module. For example, here we model a physical processor and the accompanying software layers. The PEs APPLAYER (listing 6 line 4) represents the application layer. Application is executed in spawned threads inside this PE. The PE HAL (listing 6 line7) represents the Hardware Abstraction Layer of the processor, HAL implements the service HALS functions which provide an abstract interface for the application to access low-level processor functions and mediates access to common hardware like request to/from the memory or request an update from the control plane. The HAL internally uses the processor ports to access and communicate with other modules. In figure 13 the core0, app and hal blocks represent processing elements.

5.1.3.1. Top-Level PE

In FAD, the PE top (line 17) has a special role similar to the main function in C/C++ programs. It represents the top most level model of the simulation environment in which every component is contained. It is a required part of every complete FAD model.

The simplest possible FAD model is just:

Listing 7 Minimal FAD specification

```
0:    PE top {
1:    };
```

It is analogous to the simplest possible C/C++ program:

Listing 8 Minimal C++ Program

```
0:    int main() {
1:      return 0;
2:    }
```

5.1.4. Bind - connections in FAD

The FAD binding statement is used to specify the connections between modules. The ***bind*** keyword is used within PEs to describe the connections between that PE's children. Bind statements are quite flexible. The service port of a PE instance may be bound to an instance of a PE which implements the interface, using the bind keyword (listing 5 line 14). An interface port of a PE instance can also be bound to an instance of a CE which implements the same interface, using bind (listing 6 lines 15, 20). Modules of different interface/service types cannot be bound to each other.

5.1.5. Service

The keyword ***Service***, is similar to an interface; it defines a software service type which is used for the connections between software modules. Its behavioral description is also a set of abstract functions which define the API that a software layer would export. The key difference form *interfaces* is that *services* are implemented by Processing Elements (PE), and that they represent software connections rather than hardware connections.

Conceptually, any PE implementing a service should be thought of as representing a software layer. To continue with our CPU example, we add a Hardware Abstraction Layer (HAL) service and a PE implementing it. This HAL Service (listing 6 line 3) exports functions that a processor's API would contain to access registers/hardware of the processor. Services allow for the separation of the software stacks into pure algorithmic (APIs) and pure structural functions (drivers). This allows IP reuse of blocks from other models.

5.2. FAD Platform generator

A language by itself no matter how elegant is useless if it can't be implemented. We have designed a platform generator utility (*pfpgen)* that automatically generates the equivalent SystemC code from the FAD specification. The modeling workflow is shown in figure 23.

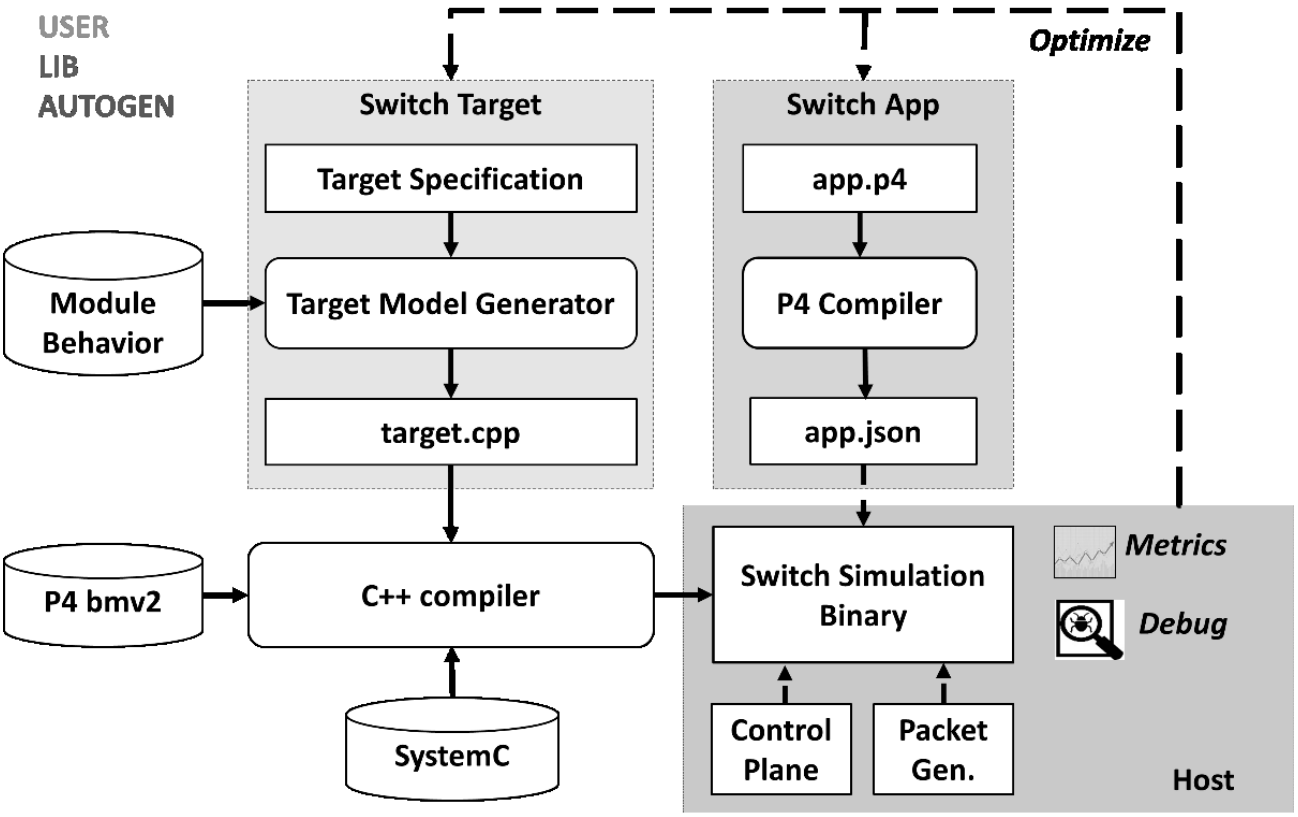

Figure 23 Modelling Workflow

The user defines the structural description of the design in FAD. The utility generates a SystemC model from FAD which the user fills out with the behavioural logic for different modules in the platform C++. The application code to be executed on the processing cores can be written

in C++ by the user. Alternately, we provide a P4 target on which the user may use the P4 compiler to run a P4 application, targeted for the platform model. The P4 compiler also generates the logic for the parsing module in the platform.

The module instances defined in the FAD specification can be configured by configuration files (module.cfg) which are a separate input to the simulation model. The configuration files contain simply a set of key-value pairs and are loaded into the simulation model, as a map, at run-time. These key-value pairs can be properties of the module like cache size, link latency; properties of the module which can be parameterized. The configuration files allow the convenience of batch simulations with multiple design options without having to recompile the simulation binary.

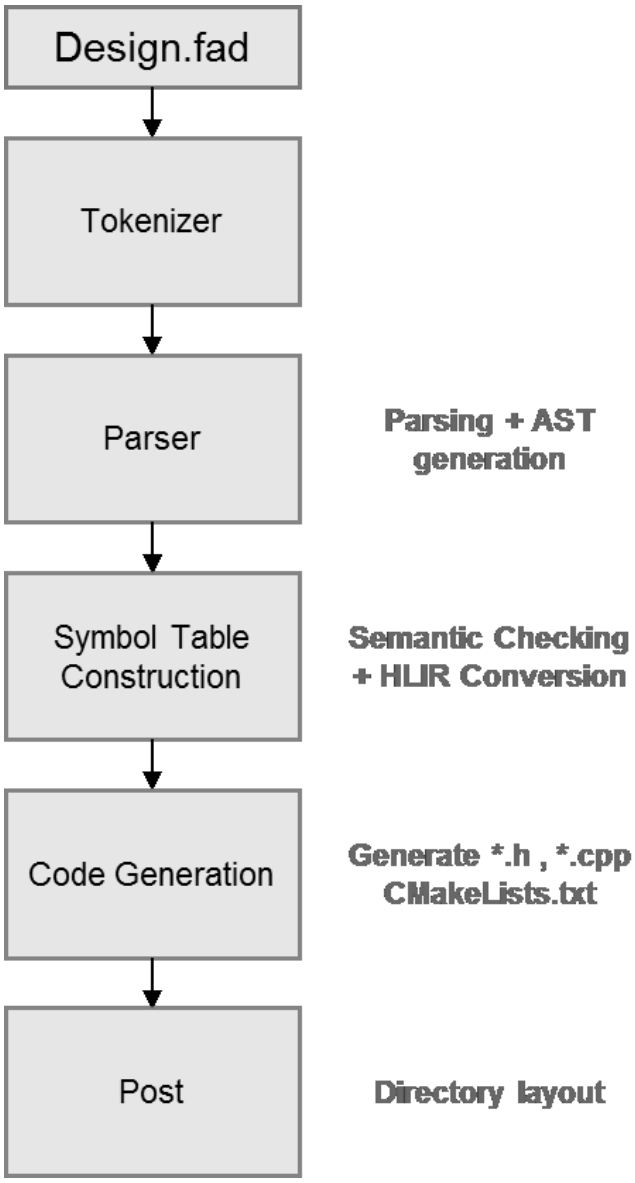

Figure 24 FAD Compiler Layout

Pfpgen's frontend is a lex-yaac compiler which parses a FAD description into an intermediate high level representation (HLIR) which is represented as an abstract syntax tree of Processing Elements. The backend of the compiler is responsible for C++/SystemC code generation. The HLIR AST is mapped to a semantic tree. The semantic tree is used to populate the

SystemC templates for FAD elements for PE's, CE's, Interfaces and Services, which the templating engine (Tenjin) uses to generate the model code.

5.2.1. Compiler Front End

The first step for any compiler is to identify the contents of the program, it needs to break the stream of characters from the file into words, phrases or tokens. The process of tokenization is performed by *lex*, a lexical analyzer. The tokenization process is governed by valid regex patterns that the language grammar allows to define the syntax of the program. The tokens are passed to a parser (*yacc*) which parses the tokens for syntaxical and semantic checking according to the language grammar expressed using BNF notation. The parser builds an abstract syntax tree representation of the program in a high level intermediate representation.

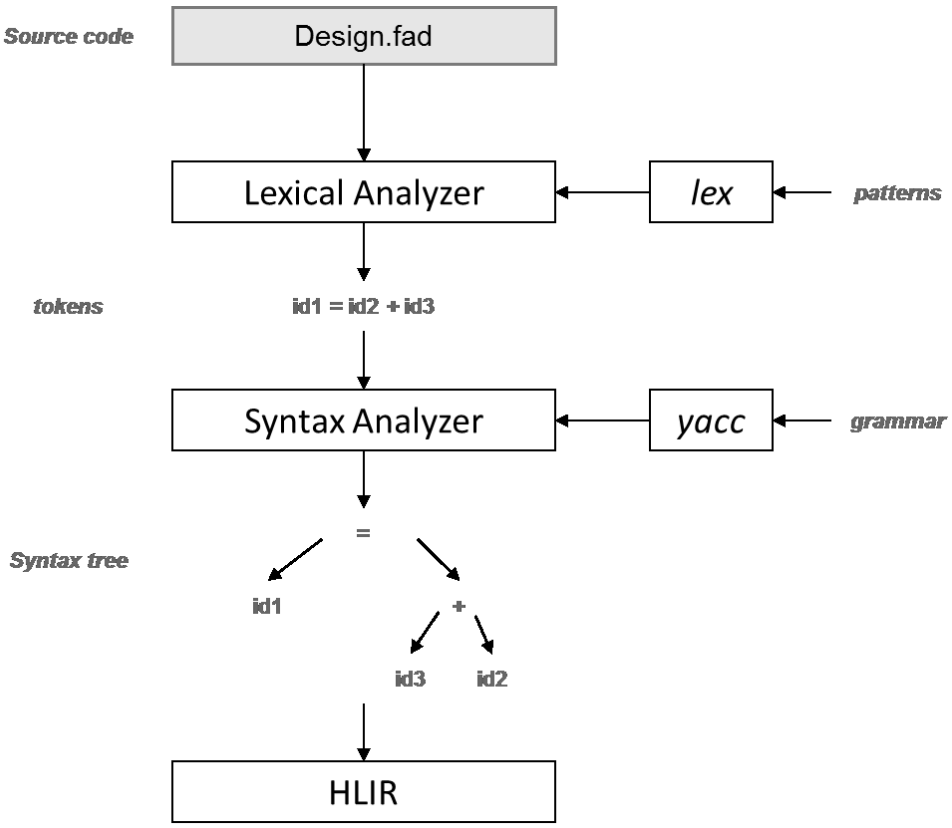

Figure 25 FAD Compiler Frontend

5.2.1.1. Lexical Analyzer

During the first phase the analyzer reads the input and converts strings in the source to tokens. The lexical analyzer returns token data structures consisting of a token types like int, string, keyword, and symbols. Each token type is qualified by a token value with a reference to the literal representation of the identifier. All valid token types are declared as list of strings to lex. Tokens are recognized using regular expressions as shown in the table below.

Table 3 FAD Lex tokenization regular expressions

TOKEN TYPE	REGEX	MATCH VALUE QUALIFER
ID	[a-zA-Z_][a-zA-Z_0-9]*	interface, service, CE, implements, PE, bind, import
STR_LITERAL	"(([^"]|\\")*)"	*
INT_LITERAL	(0|[1-9][0-9]*)	*
COMMENT	//.*	*
MCOMMENT	/*(.|\n)*?*/	*

Tokens that require no special processing (Table 3, Row 2, 3, 4, and 5) are declared using module-level variables of the form *t_TOKEN_TYPE* where TOKEN TYPE matches some string in the tokens list. Each of these token variable modules are qualified by the respective regex (Table 3 Column 2) that matches the token.

Tokens that have reserved word qualifiers require special processing are declared as module-level functions that follow the same naming convention, with a matching regex. The difference is that each function receives a value of type TOKEN TYPE which modifies it and returns the type of the instance which is then processed by the respective token module-level variable.

5.2.1.2. Parser

Python Yacc after importing the list of tokens parses these for valid expressions to build a parse tree. The context-free language grammar is specified as a set of doc strings in functions which are the parser rules. Each function takes input a single argument that is a sequence of token values matching the respective symbols of the doc string. Depending upon on the parse rule function the expression is evaluated in the body of the function and places the result in *p [0]* which forms a node of the parse tree. This is done iteratively over all tokens to form the AST.

Listing 9 FAD Parser BNF example

```
0: def p_declaration(self, p):
1:         '''declaration : interface_declaration
2:                        | service_declaration
3:                        | communication_element_declaration
4:                        | processing_element_declaration
5:                        | import_statement '''
6:         p[0] = p[1]
```

After building the AST the parser module performs semantic checking against the symbol table generated during parsing of the token to determine illegal FAD constructs by going over the tree.

5.2.2. Intermediate Representation

In order to keep the FAD compiler modular and support multiple compiler back ends for different implementations like SystemC, VHDL etc. The AST is converted to a High Level Intermediate Representation (HLIR). Since we are building the whole compiler in python the most natural format to represent HLIR is itself as a pyobject. A complete HLIR is a flattened version of

the AST in which each node is a list of attributes of the parent node. All root nodes for each construct (Processing Element, Communication Element, Service and Interface) are mapped as list of attributes of the top-level HLIR instance, as shown in the listing 10. *Self.name* (Listing 10, line 2) refers to the name of the design and *top_level_PE* is a reference to the top module in the PE hierarchy. Each child element in the list of a PE contains a reference to sub-PE's, CE's, interfaces and services of that PE.

Listing 10 FAD HLIR format

```
0: class HLIR:
1:  def __init__(self,name):
2:   self.name = name
3:   self.interfaces = []
4:   self.services   = []
5:   self.ce_members = []
6:   self.pe_members = []
7:   self.top_level_PE = None
8:   self.bindings = []
      …
```

5.2.3. Compiler Backend - SystemC Translation

The backend of the compiler is responsible for C++/SystemC code generation. The HLIR is mapped to a SystemC semantic tree of each FAD element type. The mapping of HLIR element attributes is used to populate the SystemC templates for FAD elements for PE's, CE's, Interfaces and Services, which the templating engine (*Tenjin*) uses to generate the model code. Generated code is separated into behavioural and structural directories. The user fills out logic in generated skeletons in the behavioural directory. The structural directory contains fully SystemC qualified code that define the modules and the interconnections between them. This separation allows the

compiler to overwrite files from structural changes in FAD leaving the behavioural codebase untouched.

5.2.3.1. Interface & Service

FAD Interface and Service declarations translate into SC_INTERFACEs and instances translate into SC_PORTs of those SC_INTERFACE types. Interface and services are pure abstract C++ classes, the key difference being that PE classes concertize the definition of function exported by services where vis-à-vis CE's concretize interfaces. Only Interface classes are templated so as to provide a base class (*TrType*) for passing around data structures through interfaces.

Listing 11 Interface SystemC implementation

```
                                FAD – design.fad
0:    interface <interface_name>;
                SystemC – design/behavioural/interface_name.h
0:    template <typename T>
1:    class interface_name: public sc_interface {
2:         /* Virtual functions defined by user */
3:         virtual interface_function1(...) = 0;
4:         virtual interface_function2(...) = 0;
5:    };

                                FAD – design.fad
0:    PE <pe_name> {
1:    ...
2:         <interface_type> <interface_instance_name>;
3:    ...
4:    };
                   SystemC – design/structural/pe_nameSIM.h
0:    class pe_nameSIM ... {
1:    ...
2:     public:
3:         sc_port<interface_type<TrType*>> interface_instance_name;
3:    ...
4:    };
```

5.2.3.2. Processing Element

PEs are translated into a hierarchy of SC_MODULES for HW structure. The SC_MODULES contain SC_THREADs to model logic in the PE. These are the most complex modules that are translated from FAD to a set of C++ classes. The inheritance hierarchy of a PE module is shown in the figure below.

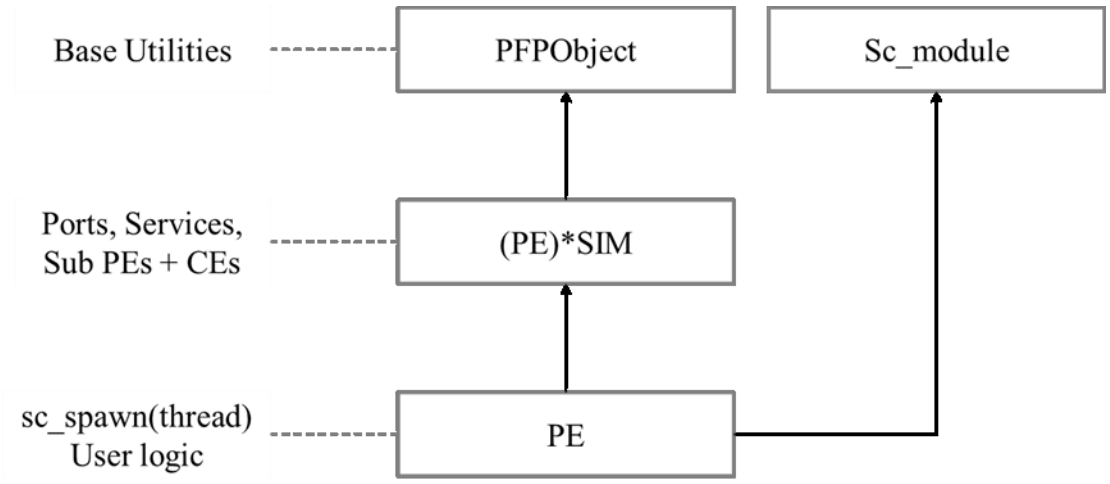

Figure 26 PE SystemC implementation

The final PE class is a complex polymorphic object that is exposed to the user, where the user describes the behavior of the module, it publically inherits from a PESIM class and an SC_MODULE. The PESIM class publically inherits from the *PFPObject* base class, which qualifies all Concrete objects in the model and provides access to common utilities like parent lookup, port inspection etc. The PESIM is a polymorphic and composited class. It houses all of the interface, sub PE and CE's instances declared inside the PE in the FAD description as public data members of the class. Sub PE and CE's are implemented as shared pointers and are initialized by the constructor of the PESIM class on the heap. If a PE implements a service, the PE class

public inherits from this service class. The user concretizes the abstract service functions inside the PE class.

The inheritance from SC_MODULE is done at the last stage because the design of SC_MODULE class itself inhibits virtual inheritance, which means an object cannot inherit from two SC_MODULES. The separation of a PE into two classes PE and PESIM is done to allow modular design. The compiler overwrites the SIM class as it contains only structural information completely deducible from the FAD file. This allows us to make structural changes without touching the behavioural codebase of the PE.

Listing 12 FAD PE SystemC Implementation

FAD – design.fad

```
PE <pe_name> implements <service_dec> {
            <child_pe> <child_pe_name>;            // PE
      <interface_dec> <interface_instance_name>;  // Interface
      ...
};
```

SystemC – design/structural/pe_nameSIM.h

```
class pe_nameSIM: public PFPObject {
...
 public:
      sc_port<interface_dec<TrType*>> interface_instance_name;
      std::shared_ptr<child_pe> child_pe_name;
};
```

SystemC – design/behavioural/pe_name.h

```
class pe_name: public pe_nameSIM, sc_module, service_dec {
...
 public:
      void init();
      virtual void Service_f1(...);
      virtual void Service_f2(...);
...
};
```

5.2.3.3. Communication Elements

Memory CEs are translated into passive transaction-level SC_MODULEs that provides communication services through the implemented interface. They follow the same inheritance scheme as PEs but there are no *SIM classes for CE's since there is no structural information inside a CE. All transactions that communication elements process inherit from the base class *TrType.* Data structures can be up casted to the required classes thus providing runtime checks for communication elements on transaction types. CE's are templated due to inheritance from interfaces they are purely header implementations residing in the behavioural directory of the model.

Listing 13 FAD CE SystemC implementation

FAD – design.fad

```
0:    interface <interface_dec>;
1:    CE <ce_name> implements <interface_dec>;
```

SystemC – design/behavioural/ce_name.h

```
0:    template <typename T>
1:    class ce_name: public interface_dec, sc_module, pfpobject {
2:          /* Virtual functions implemented by user */
3:          virtual Mem_function1(...) {
4:                ...
5:          }
6:          virtual Mem_function2(...) {
7:                ...
8:          }
9:    };
```

5.2.3.4. Bindings

Bind statements in FAD are simply translated to their equivalent sc_bind statements. Since SystemC bindings are to be completed in the elaboration phase before the simulation call the sc_binds are carried out in the constructor of the PESIM class.

Listing 14 FAD Bind statement SystemC implementation

FAD – design.fad

```
0:    PE <pe_name> {
1:     ...
2:     bind <pe_name>.<service_ins>  { <pe_name> };   // Service - PE
3:     bind <pe_name>.<interface_ins>{ <ce_name> };   // Interface - CE
4:     bind <pe_name>.<interface_ins { <ce_name> };
5:     ...
6: };
```

SystemC – design/structural/pe_nameSIM.cpp

```
0:    pe_nameSIM(…): … {
1:          ...
2:          pe_name -> service_ins.bind(*(pe_name.get()));
3:          pe_name -> interface_ins.bind(ce_name);
4:          pe_name -> interface_ins.bind(ce_name);
5:          ...
9:    }
```

5.3. Summary

In this section we presented the forwarding architecture description that can be used to succinctly describe forwarding planes and the implementation of FAD to SystemC compiler which generates complete structural code and behavioural skeletons that the user can fill in to define the behaviour of models. This distinct separation allows us to decrease the modelling effort by making quick changes in FAD to be translated to SystemC code which previously would have to be undertaken manually by the simulation team.

Chapter 6: Experimental Results

6.1. Architectural Exploration

Silicon area is expensive. Architects tread the thin line where they have to balance a variety of factors from performance to power consumption and each of these factors is affected by the silicon die space they allocate to each subsystem. Giving larger die space to more memory vs. placing an extra core can be a difficult choice, the extra core means more processing power is available but this might push the thermal and power budget to its max TDP constraint for the design, however having an extra core will be for nothing if the memory banks are too small and the cores mostly have to wait on I/O for them to flush for data fetches because they can't manage to fill the processing pipeline. All interesting questions but are difficult to answer as each variation on the architecture requires a model. This is the point at which the architect can lament or rejoice in the choice of their tool chain and abstraction levels of their models.

6.1.1. Evaluation Criteria

We created variations of the NPU architecture model shown in figure 4, where we vary the number of clusters and on-chip memory per cluster to reflect design trade-offs that may occur in limited die space or to fit certain power budgets. Since only the structure is changing for the NPU we created FADs for each variation and used *pfpgen* to generate the structural code, where behavioural code for all modules remained the same across architectures.

In addition to that we compared it to a high-level model of the RMT describe in section 3. Finally, we also developed a soft-switch implementation in SystemC, based on the generated code from

the P4 compiler. A 2.7 GHz Intel dual-core machine with 8GB RAM was used as our simulation host.

To assess the value addition of models we use simulation speed as a metric. In particular, we demonstrate that even complex NPU architectures with a large number of cores can be simulated with realistic applications and traffic in seconds. We also ascertain the usability of models in design space exploration of different architectures using average packet latency as a common metric for the same application.

6.1.2. Test Cases

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). Simple router is a simple L2/L3 application which first performs a longest-prefix match on the destination IPv4 address and uses this to set the egress port and next-hop address of the packet, additionally decrementing the Ipv4 TTL field. The next stage performs an exact match on the next-hop address from the previous stage, and the result is used to rewrite the Ethernet destination MAC address. Next an exact match is 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 are updated during deparsing. The match table sizes were set to 2048.

Models were simulated with 5000 pseudo-randomly generated packets of size 1KB each. The packet trace included all unique IP addresses in the match tables at least once in order to force

the search algorithm to access every node in the tries in memory. The packet generator generated packets at a rate of 1 GPPS at the ingress port.

6.1.3. Test Models

Based on the NPU architecture shown in Figure 4, we created six NPU models with 1, 2, 4, 8, 12 and 16 processing clusters. Each cluster consisted of four 4-way hardware multi-threaded processing cores. Each cluster had 1 eDRAM block with single cycle access. There was only one 256 MB off-chip DRAM with access latency of 10 cycles. The sample NPU FAD is attached in Appendix-A. The scheduling policy is round-robin over the clusters. The second test model is the reconfigurable pipeline model described in section 3 based of the RMT architecture [5].

The P4 soft-switch was modeled as a host-compiled C code inside a single PE (SC_MODULE), 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 only useful for functional validation of the P4 program against the other architectures.

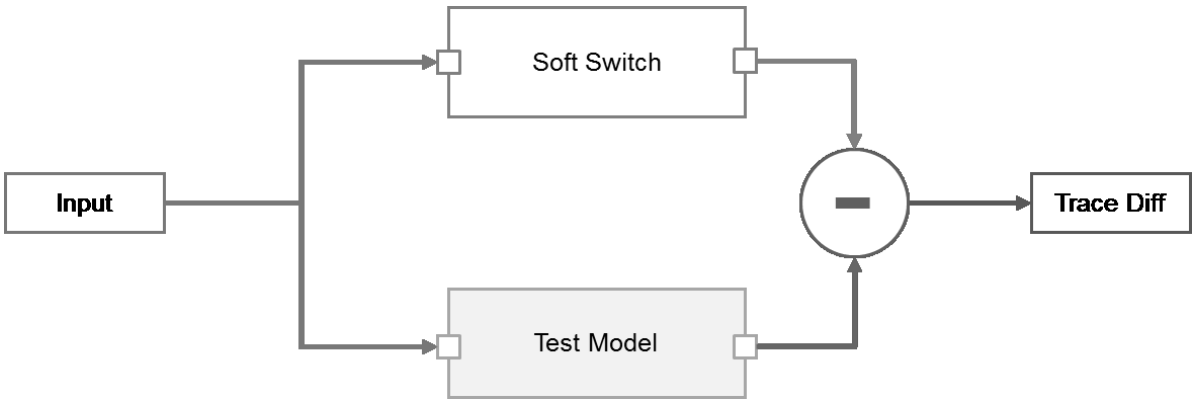

Figure 27 Functional Validation of Models

6.1.4. Results

Table 4 Experimental Results

Design	Processing	On-chip memory model capacity	Simulation Time (s)	Latency
soft-switch	dual-core host	host memory	0.456	N/A
RMT	32 stages	TCAM + hash	0.339	102 NS
NPU-1	4 cores	64K eDRAM	8.438	23.8 µS
NPU-2	8 cores	32K eDRAM	8.639	11.3 µS
NPU-4	16 cores	16K eDRAM	8.947	5.1 µS
NPU-8	32 cores	8K eDRAM	10.41	1.9 µS
NPU-12	48 cores	6K eDRAM	13.501	**0.9 µS**
NPU-16	64 cores	4k eDRAM	18.967	24.0 µS

Table 4 presents our experimental results. The NPU design space exploration consisted of increasing the number of cores, while reducing the on-chip eDRAM capacity to fit within the chip area budget. The total eDRAM capacity (in Table 4, Column 3) is distributed equally across the clusters. For instance, in NPU-8, each of the 8 clusters have 1Kb eDRAM for a total eDRAM capacity of 8Kb. The tries corresponding to the match-tables were copied into each of the identical eDRAMs in the clusters by the control plane agent. The structures spilled over into the singleton off-chip DRAM if needed. Therefore, all application threads running on a core, in a given cluster, could access only the cluster's eDRAM or the off-chip DRAM.

The SystemC model generation from the FAD using *pfpgen* took only on average **0.378** seconds and was consistent which shows little correlation to the complexity of the design. However, the reported packet latency and simulation speed showed interesting trends as discussed below.

6.1.5. Simulation Speed

The soft-switch implementation derived from the P4 example processed all the packets under a second on the host. The RMT model simulation was faster than the soft-switch simulation because of the underlying parallelism of the pipeline stage models and the mapping of SystemC threads to the dual-core CPU.

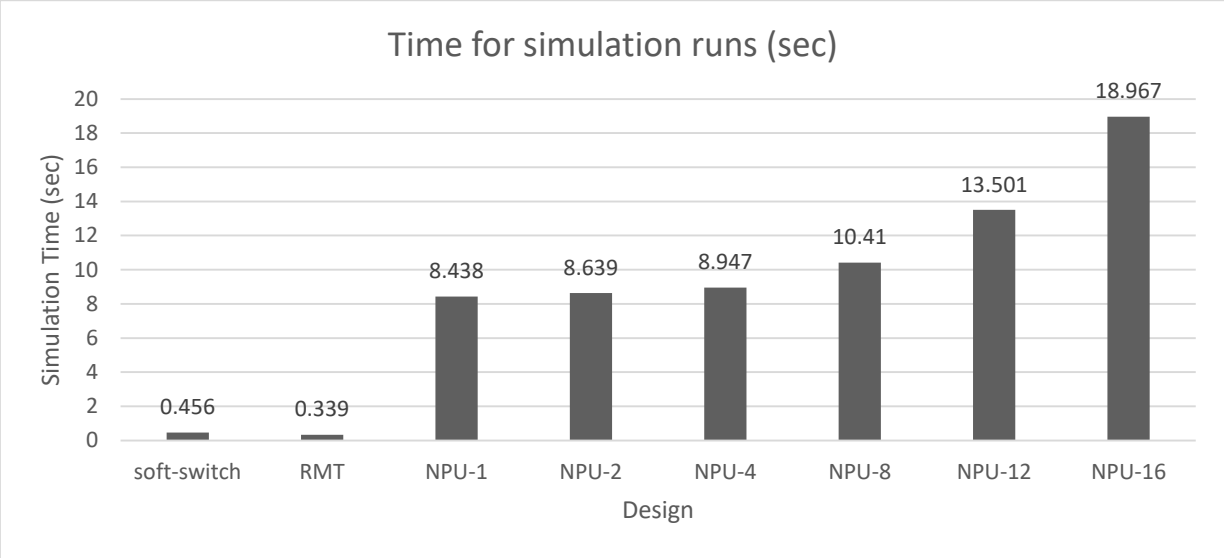

Figure 28 Wall clock time for Simulation runs

The simulation speed for the NPU models decreases with the increasing number of cores. This is to be expected since the processing of the same 5000 packets in different models is being executed on more threads in each successive design as scale from 1 4x4 cluster to 16 4x4 clusters which is an increase of 16x in terms of application threads in the simulation on the host. As such there are more context switches between the simulation threads leading to slower simulation.

Compared to the untimed soft-switch execution, simulating a 64 core NPU with approximate level timing results in a slowdown of only 40X. While this may seem like a huge

factor, the absolute simulation times are still in the order of few seconds, which is acceptable considering the depth of the architectural details captured in the NPU model. The simulation speed can be increased by using a more powerful host or utilizing a parallelized or a GPU-accelerated SystemC simulation kernel.

6.1.6. Average Packet Latency Estimation

The packet latency for the RMT is constant because the parsing and deparsing stages take exactly 3 cycles each due to the fact there are only three headers (Ethernet, IPV4 and TCP) in P4 test program, while each of the 32 match action stages take 3 cycles, since the action processor is not as detailed modeled as the processing cores of the NPUs it leads to an overall latency of 102 ns. If the P4 program includes headers that are more complex there will be additional delay at the parsers and deparser, also more hash conflicts during match stages would factor into a greater delay.

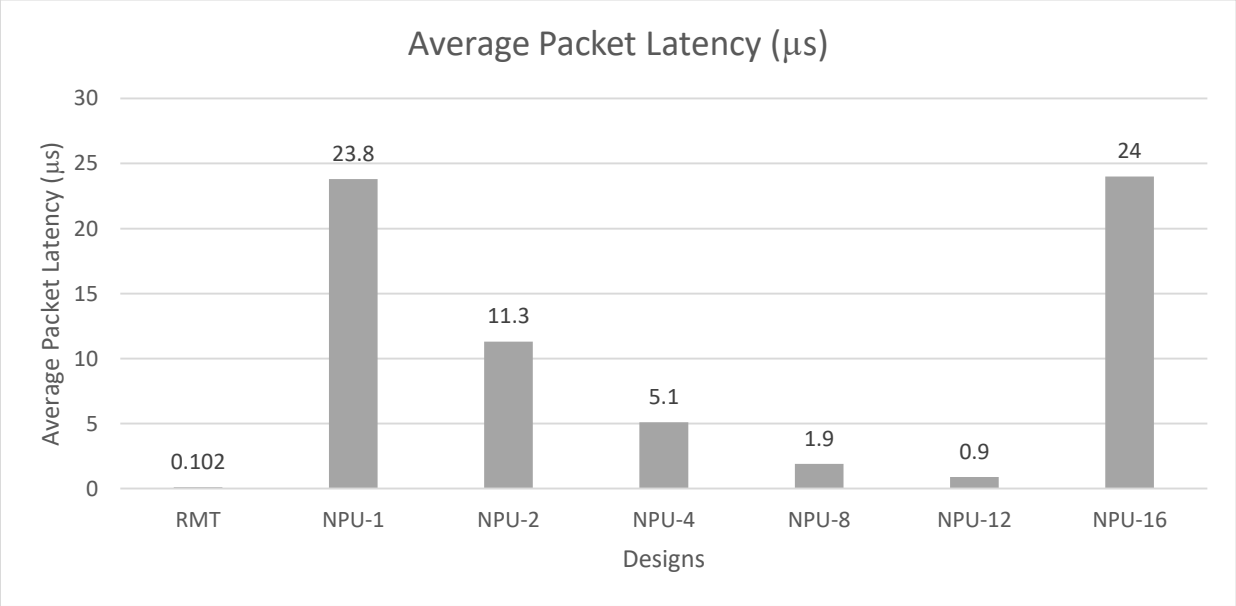

Figure 29 Average Packet latency of test designs

For the NPUs, the average packet latency decreases as we increase the number of clusters from 1 to 12. Although we are simultaneously decreasing the total eDRAM size, the corresponding tries for lookup still fit in the on-chip memory, given the modest table size of 2048 entries of our synthetic example. However, when we increase the number of clusters to 16 and reduce the per-cluster eDRAM size to only 256B, the tries spill over into the off chip DRAM which has a 10x latency compared to the single cycle eDRAM access. Hence, there are a significant number of slow DRAM accesses in the NPU-16 design which not only arise from slower access times but also because processing clusters are contending with each other for access to the DRAM, which limits to how much of the workload can processed concurrently. As a result, we notice a marked increase in average packet latency in NPU-16, undoing all the benefits of higher parallelism.

The unexpected trend that emerges from the graph above is the factor of improvement in average packet latency when we increase the number of clusters. A naïve assumption would be that doubling the number of cores would result in at most 2X average latency reduction. However, recall that the scheduler at the NPU level assigns the packets to clusters in a round robin fashion. Since at the ingress packets are arriving in constant time, the packet inter-arrival delay at a cluster increases with the increase in the number of clusters. As a result, the cores inside the clusters have fewer conflicts on the shared eDRAM, leading to faster packet processing within the cluster. Therefore, we observe a super linear improvement in packet processing latency with respect to the number of clusters i.e. until the tables can no longer fit in the on-chip memory.

6.1.7. Accuracy

The models were functionally verified against the P4 soft switch model but accuracy of the underlying model is an important metric for any simulator framework, since we are not just

modelling one small part like the processor of the system. Accuracy of the full system simulation has to come from the timing of components in the system model both fixed function and dynamic modules. In this case, it is up to the designer to provide timing annotations in components as part of the behavioural module logic. In our experiments, we have used timing data for memories from component datasheets.

6.2. Algorithm Evaluation

Algorithms are evaluated by the resources they use, such as computational complexity and storage requirements. An algorithm can be compute or memory intensive or even both. Performance of such algorithms depends highly on the architecture that they are being executed on to leverage maximum performance. Hardware and software development cycles now are becoming more are more concurrent where most target hardware is often available late in the stages of software development. At this point bulk of the algorithm such as data structure choice has already been fleshed out but development in early stages give no indication of performance on the target architecture.

One of the core functions of forwarding planes is Layer 2 / Layer 3 packet switching. Forwarding information is stored in lookup tables called Forwarding Information Bases (FIBs). FIBs are constructed from routing tables which define how nodes in a network are connected to each other. Where routing tables are optimized for efficient updating of nodes, FIBs are optimized for fast lookup of destination addresses since lookups are performed for every processed packet. For a given datagram in L2 switching for ethernet headers exact matches are performed. For L3 next hop IP addresses are determined using Longest Matching Prefix (LPM) to select an entry from the forwarding table. The LPM algorithm is used for IP lookups since it is a best-effort

protocol and each entry in a forwarding table may specify a sub-network, a destination address can match one or more route. The entry with the longest subnet mask i.e. where the largest number of leading address bits of the destination address match an entry in FIB is called the longest prefix match. This mechanism of examining entries in order of leading bits causes LPM to be a very memory intensive operation. The natural choice to represent FIBs are tries or trees, since it is an ordered data structure that can be easily sorted and searched through. FIBs can grow to be very large and lookups have to be fast so *LPM* algorithms have to minimize memory accesses and memory size.

6.2.1. Test Environment

To demonstrate how PFPSIM can be used for software evaluation we first define our base testing model, from the previous case we use our base NPU model which has 8 processing clusters. Each cluster has four eDRAMs of 2Mb each. Each eDRAM has a single cycle access latency. The cluster on chip network connects them to 8 cores where each core is 4 way threaded.

The sample P4 program is a variant of *simple_router* example of P4. This is a L2/L3 switch which has three match-action tables and two header types (Ethernet and IPv4). The first match action stage performs a longest prefix match on the IPv4 destination address for the next hop this is used to set the next-hop address and egress port of the of the packet. It also decrements the TTL field of the IPv4 header by one. The second match-action stage performs an exact match on the next_hop IPv4 address set in the previous stage, the result rewrites the Ethernet MAC address. The third table finally performs an exact match for the egress port determined in the first match table to rewrite the source Ethernet MAC address. All IP checksums are updated accordingly. The P4 application is modeled as a host-compiled code in SC_THREADs executing in the Application

Layer PE in the NPU Cores. Models were simulated with 5000 pseudo-randomly generated packets of size 3KB each. The packet trace included all unique IP addresses in the match tables at least once in order to force the search algorithm to access every node in the tries in memory. The packet generator generated packets at a rate of 1 GPPS at the ingress port

6.2.2. Test Cases

The performance of *LPM* searches in the P4 application is highly dependent on the data structure used. The P4 runtime has been modified to use customized trie search structures which use TLM-VAR to redirect memory access to the memory models. We will be testing the performance of Prefix Tree, LC Trie, and Multibit-Variable Stride Trie. These algorithms are all of O(n) complexity. A Prefix Tree implementation is just a binary-radix tree where the nodes of the tree are the leading bits of the prefix from the root node, each node either stores the pointer to the default action or next hop if the node is terminating address bit for a prefix entry in the forwarding table. The worst case for IPv4 (32-bit addresses) is that it needs to add 32 nodes which increases storage complexity by 32xNxS for *S* entries in the tree. The search and update complexity is *O(n)* where *n* is the length of prefix and storage complexity is *O(nW)*.

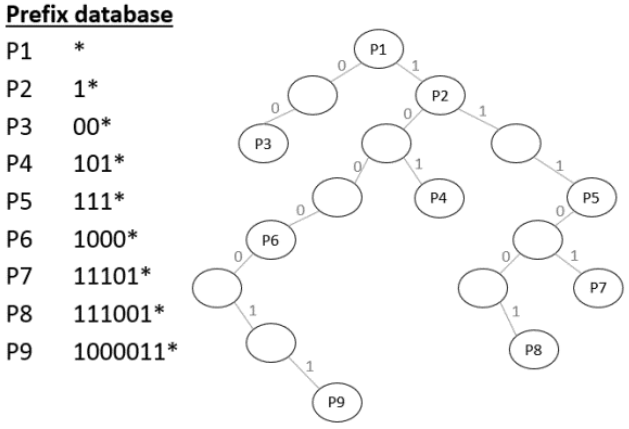

Figure 30 Prefix Tree representation

An LC-Trie is a compressed version of a prefix tree. Nodes that do not contain a next-hop and only has a child are removed to form a shorter part from the root node. Since nodes are removed, each node stores the skipped bits in the prefix. To construct a LC-Trie the algorithm starts with a disjoint-prefix unibit trie (only leaf nodes contain prefixes). Fig. 31 shows three different tries: (a) unibit trie where prefixes at leaf nodes; (b) path-compressed trie; and (c) LC-trie. The first three levels of the path-compressed trie that form a full sub-trie are converted to a single-level 3-bit sub-trie in the LC-trie. The LC-trie needs only three levels instead of the six required in the path-compressed trie. For a search prefix it is compared to the skip bit indicating the index of the bit to be tested to decide which path to take out of that node. Thus, we jump directly to the bit where a significant decision is to be made, bypassing the bit comparisons at nodes where all the keys in the subtree have the same bit value. The compression reduces the height of the tree, the lookup is still *O(n)* but for this *n* in case of LC Trie is always smaller or in the worst case of no compression equal to the Prefix Tree. Since nodes are eliminated memory space is at most *2N-1* where N is number of leaves in the prefix tree. Since paths are compressed and internal nodes are removed, the space complexity compared to prefix tree becomes just *O(N)* and is independent of the worst case scenario *W*.

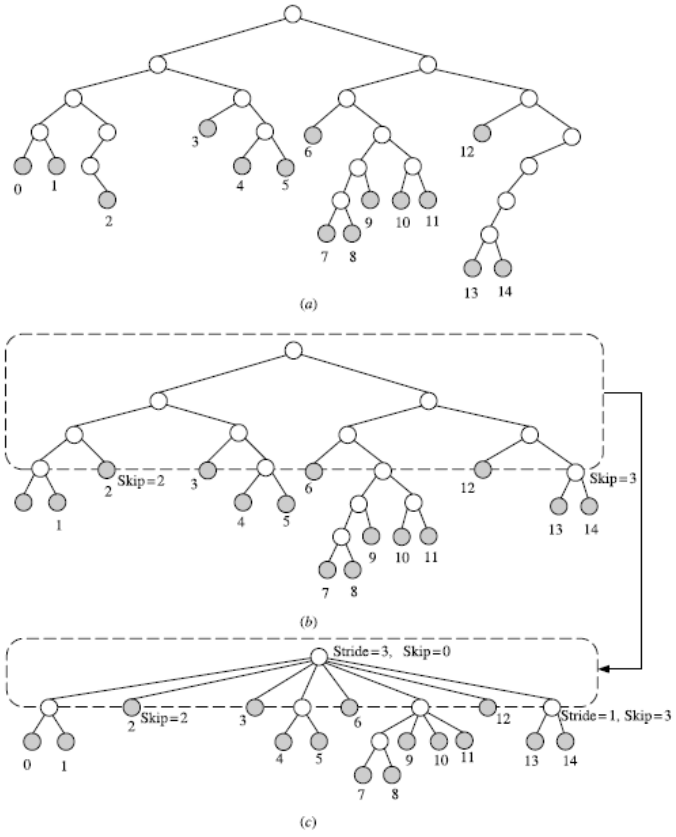

Figure 31 LC Trie representation

Although when a prefix tree is full i.e. each node has two children there will be no compression. Since internal nodes that have single child nodes are removed the LC trie cannot be updated for new entries it has to be re-constructed.

Searches in Prefix Trees and LC-Trie still happen one bit at a time. In the worst case it would take 32 memory access for an IPv4 address to match. If the memory access time is 1 ns, the lookup operation will consume 32 ns. This implies a maximum forwarding speed of 31.25 million packets per second (mpps) (1/32ns). If we assume minimum sized 40-byte IP packets which is a TCP-acknowledgment packets, this can support links speed of at most 1.3 Gbps. The processing rate can be improved by reducing search time, For example, if we examine 4 bits at a time, the lookup will finish in 8 memory accesses as compared to 32 memory accesses in a Prefix tree. In the multibit trie structure, each node has a record of 2^{stride} entries and each has two fields: one for the stored prefix and one for a pointer that points to the next child node. If all the nodes at the same level have the same stride size, we call it a fixed stride; otherwise, it is a variable stride. The number of memory accesses needed depends on the number of levels or the height of the trie. Number of levels = W/K where W in the worst case is 32 and K is the stride size. As we increase the stride size the number of levels will decrease and the number of memory access will also decrease. However, memory space consumed will be larger. Therefore, when choosing stride size a trade-off happens between the memory accesses and space requirements. For example, if the maximum lookup speed should be 30ns and the designer is presented with memory chips at a cost of 10ns for access time, then the number of levels needs to be at max 3. Since choosing the optimal stride size of each level in order to minimize the memory space varies and is highly dependent on the prefixes database, this is effectively a brute-force NP complete problem and outside the scope

of this discussion; stride lengths are usually pre-computed off-shelf from RIBs when extracting FIBs.

Lookup is performed in strides of *k* bits by choosing leading address bits equal to the stride length of the node, the lookup then follows the pointers to next nodes for matches in the bucket as show in figure 32

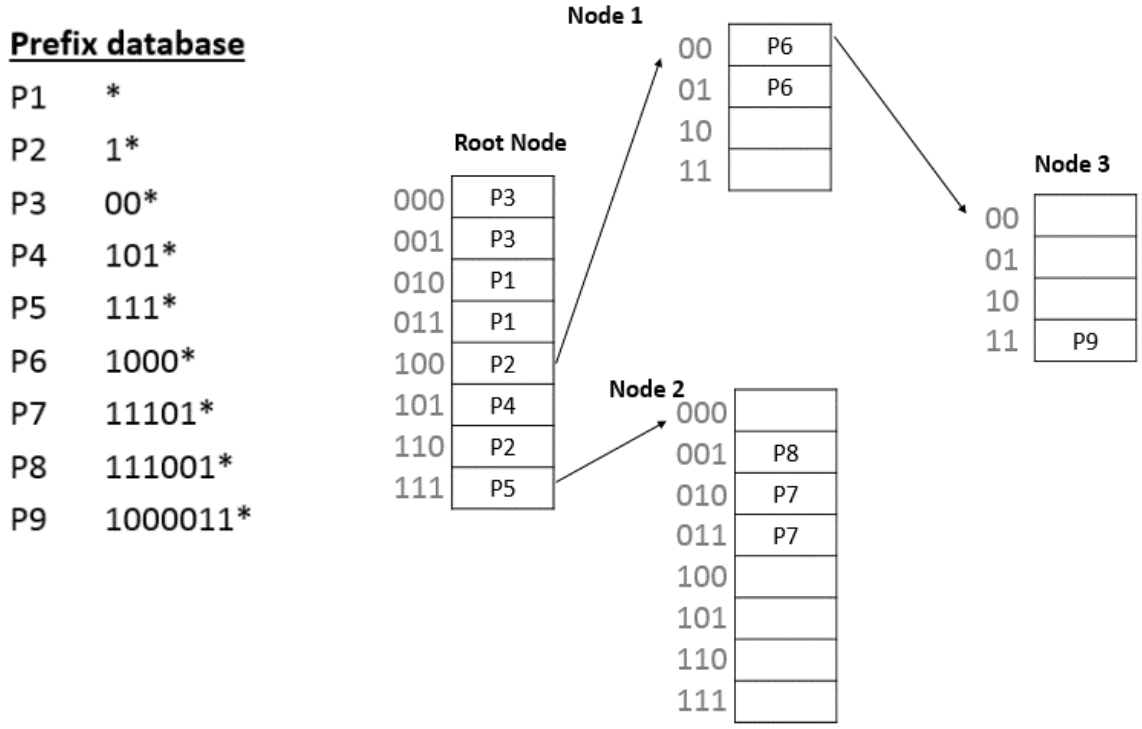

Figure 32 MultiBit Trie

The main advantage of the k-bit trie is that it improves the lookup by k times. The disadvantage is that a large space is required. For example, note the waste of space in Node 1, Node 2, and Node 3. The lookup complexity is the number of bits in the prefix divided by k bits, O(W/k). An update requires a search through W/k lookup iterations plus access to each child node (2^k). The update

complexity is $O(W/k + 2^k)$. In the worst case, each prefix would need an entire path of length (W/k) and each node would have 2^k entries. The space complexity would then be $O((2^k * N * W)/k)$.

6.2.3. Results

Table 5 Trie Performance

DATA STRUCTURE	PACKET LATENCY (NS)	BUILD TIME (US)	TRIE SIZE (KB)	AVERAGE SEARCH TIME (NS)	MAX SEARCH TIME (NS)	MIN SEARCH TIME (NS)
PREFIXTREE	430.8614844	457693	37.5521	225.8	256.99	17.93
LCTRIE	128.3789063	85697	2.7316	41.87797	87	19
MULTIBIT VARIABLE TRIE	122.3046875	69886532	1122.8434	37.0296875	42	36

Table 5 presents our experimental results for different tries. A modest table size of 1024 entries was used to construct all tries for each run. The tries corresponding to the match-tables were copied into each of the identical eDRAMs in the clusters by the control plane agent. The eDRAM size of the model was configured to be big enough that such that all tries data structures fit on the on-chip cluster memory so that we don't see contention from clusters on the off-chip memory. Therefore, all application threads running on a core, in a given cluster, could access only the cluster's eDRAM for lookups. The reported metrics show interesting trends as discussed in the next section.

6.2.4. Trie Performance

Prefix Tree noticeably has the worst performance in term of packet latency, as we can see in table 5 where LC and Multibit Variable Stride are head to head. Although all algorithms are O(n) in case of Prefix Tree it has search time is directly proportional to the length of the prefix.

The reason that the LC Tries search times approach the Multibit Variable Trie is that for this data-set there are sufficient empty nodes in the tree that it can compress, which is evident from the construction time of LC (85697 us) which is significantly smaller than that of the Prefix Tree (457693us). This shows the highly data dependent nature of LC.

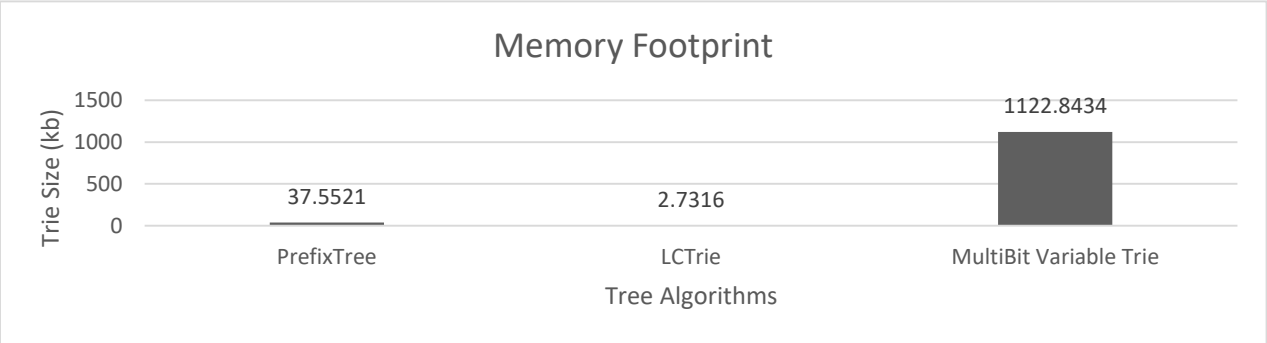

Figure 33 Trie Memory Footprint

Although MultiBit trie has lowest search times of them all, it has the highest construction time and memory footprint of the either of the tries. The reason Multibit trie takes a lot of space and build time comes from increased allocation overhead which results in a lot of space wasted for nodes that do not have next hop based on the strides lengths chosen. The implementation can be further optimized to be a similar LC-Trie like where all actions are stored in a simple stack like structure and the tree has pointers to this stack instead of allocating nodes for all slots in the stride lengths.

6.2.5. Lookup performance

Prefix tree as expected as has the worst lookup performance, with the MultiBit and LC Trie close to each other. Average packet lookup does not give us the complete picture about the performance of the algorithm. As listed in table 5, the variation of the lookup time for the LC Trie is much greater that the Multibit Variable Stride Trie and that is aptly reflected in figure 34**.** On

the x-axis it charts groups of prefixes from the same subnet and these groups are sorted in ascending order the leading zeros. We see the expected trend shown by the prefix tree as length of prefixes increase. Also we observe that the search time of LC trie increases as the length of prefixes increase which it can no longer compress as efficiently as prefixes having leading zero bits. The most interesting trend is the behaviour of Multibit Trie across the range of lookups which performs with a relatively stable lookup time. This lookup time however comes at the cost of higher memory usage as discussed in section 6.2.2.

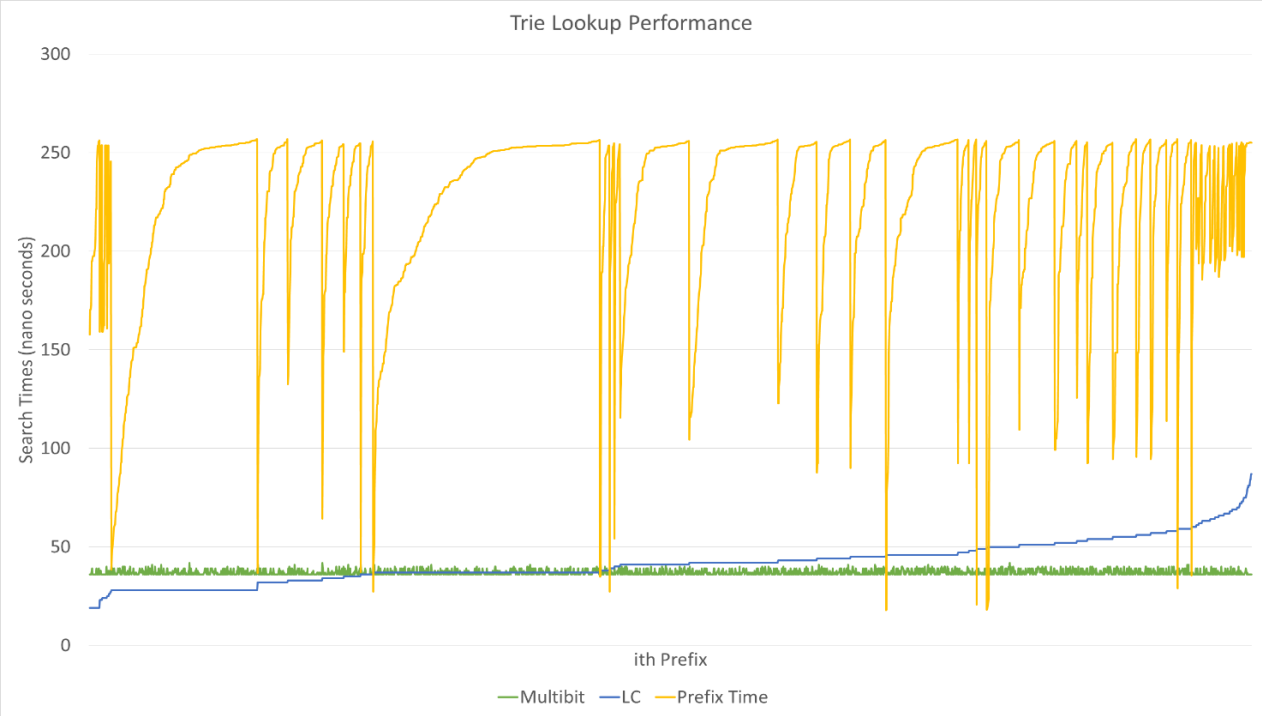

Figure 34 Trie Search Times

Chapter 7: Conclusion and Future Work

In this thesis, we presented the design and implementation of a simulation framework for programmable forwarding systems such as NPUs and match action table pipelines for early modeling, validation and performance analysis. The major contributions of the thesis are listed below:

- We demonstrated the ease and flexibility of the Forwarding Architecture Description language which provides constructs for concisely describing a platform's hardware-software structure without having to create a detailed model in SystemC.
- We implemented a compiler for FAD that generates C++ 11 compliant SystemC models with complete structural SystemC code and behavioural skeletons.
- We proposed and demonstrated a compile time solution using a base class layer for redirecting memory accesses to memory models for host compiled simulations, all the while managing to avoid the expensive decode step involved in instruction set simulators.
- The simulation framework although targeted towards modelling forwarding planes, FAD provides generic enough constructs that be used to generate boilerplate code for use in any EDA workflow.

Some of the work that is planned to be done in the future is the following:

- Introduce more constructs to the FAD language to make it easier to express structural connections.
- Perform accuracy comparisons against cycle-accurate simulations.

- Develop more tools that allow for more detailed annotation of memory accesses and support memory layouts for more complex data structures.
- Performance improvements in the simulation framework for faster results and validation.

Chapter 8: References

[1] N. Mckeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, and S. Louis, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69, 2008.

[2] H. J. Chao and B. Liu, *High Performance Switches and Routers*. 2006.

[3] S. Abdi, U. Aftab, G. Bailey, B. Boughzala, F. Dewal, S. Parsazad, and E. Tremblay, "PFPSim: A Programmable Forwarding Plane Simulator," *Proc. 2016 Symp. Archit. Netw. Commun. Syst. - ANCS '16*, pp. 55–60, 2016.

[4] M. A. Franklin, P. Crowley, H. Hadimioglu, and P. Onufryk, *Network processor design. Vol. 2, Issues and practices*. Academic, 2003.

[5] P. Bosshart, G. Gibb, H. Kim, G. Varghese, N. Mckeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," *Acm Sigcomm*, pp. 99–110, 2013.

[6] "WHITE PAPER Intel® Ethernet Switch FM6000 Series -Software Defined Networking."

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

[8] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. - Conex. '12*, p. 253, 2012.

[9] M. Gupta, J. Sommers, and P. Barford, "Fast, accurate simulation for SDN prototyping," *Proc. Second ACM SIGCOMM Work. Hot Top. Softw. Defin. Netw. - HotSDN '13*, p. 31, 2013.

[10] J. H. Ahn, S. Li, S. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *ISPASS 2013 - IEEE International Symposium on Performance Analysis of Systems and Software*, 2013, pp. 74–85.

[11] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," *ReCoSoC 2012 - 7th Int. Work. Reconfigurable Commun. Syst. Proc.*, 2012.

[12] "OVPSIM." [Online]. Available: http://www.ovpworld.org/welcome.

[13] B. Heller, "OpenFlow Switch Specification," *Current*, vol. 0, pp. 1–36, 2009.

[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. Mckeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *arXiv:1312.1719v3 [cs.NI]*, vol. 44, no. 3, p. 8, 2014.

[15] H. Song, "Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane," *HotSDN '13*, pp. 127–132, 2013.

[16] “Best Practices for using Open vSwitch with XenServer v6.0 - 6.2.0 Service Pack 1.”

[17] Petra Jorgenson, “Virtual Networking 101: Understanding VMware Networking,” 2012.

[18] “Open vSwitch on Hyper-V.” [Online]. Available: https://cloudbase.it/open-vswitch-on-hyper-v/.

[19] “Open vSwitch.” [Online]. Available: http://openvswitch.org/.

[20] T. Austin, E. Larson, and D. Ernest, “SimpleScalar: An infrastructure for computer system modeling,” *Computer (Long. Beach. Calif).*, vol. 35, no. 2, pp. 59–67, 2002.

[21] “Windriver.” [Online]. Available: www.windriver.com/products/simics.

[22] M. T. Yourst, “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *ISPASS 2007: IEEE International Symposium on Performance Analysis of Systems and Software*, 2007, pp. 23–34.

[23] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. D. Hill, D. A. D. A. A. Wood, B. Beckmann, G. Black, S. K. S. K. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, A. Basil, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. M. D. D. Hill, and D. A. D. A. A. Wood, “The gem5 Simulator,” *Comput. Archit. News*, vol. 39, no. 2, p. 1, 2011.

[24] F. Bellard, “QEMU, a fast and portable dynamic translator.,” *USENIX Annu. Tech. Conf. Free.*, 2005.

[25] D. Thach, Y. Tamiya, S. Kuwamura, and A. Ike, “Fast cycle estimation methodology for instruction-level emulator,” in *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2012, pp. 248–251.

[26] J. Sanguinetti and D. Pursley, “High-Level Modeling and Hardware Implementation with General-Purpose Languages and High-level Synthesis,” *Design*, no. April, pp. 1–5, 2002.

[27] L. Cai and D. D. Gajski, “Transaction level modeling: an overview,” *First IEEE/ACM/IFIP Int. Conf. Hardware/ Softw. Codesign Syst. Synth. (IEEE Cat. No.03TH8721)*, pp. 19–24, 2003.

[28] A. Rose, S. Swan, J. Pierce, and J.-M. Fernandez, “Transaction Level Modeling in SystemC.”

[29] “C++ Memory Model.” [Online]. Available: http://en.cppreference.com/w/cpp/language/memory_model.

[30] Danny Kalev, “C++ The Object Model,” 2003.

[31] R. Smith, “Working Draft, Standard for Programming Language C ++,” 2014.

[32] “C++ new expression.” [Online]. Available: http://en.cppreference.com/w/cpp/language/new.

[33] “C++ new, new[] operator.” [Online]. Available: http://en.cppreference.com/w/cpp/memory/new/operator_new.

[34] “RAII C++.” [Online]. Available: http://en.cppreference.com/w/cpp/language/raii.

[35] “C++ concepts: PODType.” [Online]. Available: http://en.cppreference.com/w/cpp/concept/PODType.

[36] Bjarne Stroustrup, “C++ overloading for primitive types.” [Online]. Available: https://isocpp.org/wiki/faq/intrinsic-types#intrinsics-and-operator-overloading.

[37] “sizeof.” [Online]. Available: http://en.cppreference.com/w/cpp/language/sizeof.

[38] M. A. Ellis and B. Stroustrup, “The Annotated C++ Reference Manual.”

[39] Danny Kalev, “C++ The object model: Multiple Inheritance,” 2003.

[40] “P4.” [Online]. Available: http://p4.org/.

[41] “Virtual networking in Linux.” [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-virtual-networking/.

Chapter 9: Appendix – A

NPU FAD

Eight Processing Clusters with Eight Cores and 4 eDRAMs each.

```
interface QueueRdI, QueueWrI, MemI;
service  HalS, ControlPlaneAgentS, ControlPlaneAgentHalS;
CE Queue("QueueConfig.cfg") implements QueueRdI, QueueWrI;
CE Memory implements MemI;

PE Router("RouterConfig.cfg") {
  QueueRdI ocn_rd_if[];
  QueueWrI ocn_wr_if[];
};

PE MemoryController {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
  MemI memory_if;
};

PE MemoryManager {
  QueueRdI rd_if;
  QueueWrI wr_if;
};

PE Mem("MemoryConfig.cfg") {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
  Memory tlm_memory;  //CE
  MemoryController memory_controller;  //PE

  bind memory_controller.memory_if {tlm_memory};
  bind memory_controller.ocn_rd_if {ocn_rd_if};
  bind memory_controller.ocn_wr_if {ocn_wr_if};
};
PE Splitter {
  QueueRdI ingress;
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
};
PE Parser("ParserConfig.cfg") {
```

```
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
};
PE Scheduler("SchedularConfig.cfg") {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
};
PE ControlPlane {
  ControlPlaneAgentS cpa;
};
PE ControlPlaneAgentHAL implements ControlPlaneAgentHalS{
    QueueRdI ocn_rd_if;
    QueueWrI ocn_wr_if;
};
PE ControlPlaneAgent implements ControlPlaneAgentS {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
    ControlPlaneAgentHAL cpagenthal;
    bind cpagenthal.ocn_rd_if{ocn_rd_if};
    bind cpagenthal.ocn_wr_if{ocn_wr_if};
};

PE HAL implements HalS{
  QueueRdI cluster_local_switch_rd_if;
  QueueWrI cluster_local_switch_wr_if;
};

PE ApplicationLayer {
  HalS  halport;
};

PE Core("CoreConfig.cfg") {
  QueueRdI  cluster_local_switch_rd_if;
  QueueWrI  cluster_local_switch_wr_if;

  HAL hal;
  ApplicationLayer applayer;
  bind applayer.halport {hal};
  bind hal.cluster_local_switch_rd_if
{cluster_local_switch_rd_if};
  bind hal.cluster_local_switch_wr_if
{cluster_local_switch_wr_if};
};

PE ClusterScheduler("ClusterSchedulerBaseConfig.cfg") {
  QueueRdI cluster_local_switch_rd_if;
  QueueWrI cluster_local_switch_wr_if;
```

```
};

PE Cluster("ClusterConfig.cfg") {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;

  Core core("Core.cfg")[8];

  ClusterScheduler cluster_scheduler;  //  Cluster Level
Scheduler for cores

  Mem edram_0_mem("memory.cfg"),
      edram_1_mem("memory.cfg"),
      edram_2_mem("memory.cfg"),
      edram_3_mem("memory.cfg");

  Router cluster_local_switch;

  Queue cluster_local_link_rd_channel[13];

  bind cluster_local_switch.ocn_rd_if {
    cluster_local_link_rd_channel[0],
    cluster_local_link_rd_channel[1],
    cluster_local_link_rd_channel[2],
    cluster_local_link_rd_channel[3],
    cluster_local_link_rd_channel[4],
    cluster_local_link_rd_channel[5],
    cluster_local_link_rd_channel[6],
    cluster_local_link_rd_channel[7],
    cluster_local_link_rd_channel[8],
    cluster_local_link_rd_channel[9],
    cluster_local_link_rd_channel[10],
    cluster_local_link_rd_channel[11],
    cluster_local_link_rd_channel[12],
    ocn_rd_if};
  bind core[0].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[0]};
  bind core[1].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[1]};
  bind core[2].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[2]};
  bind core[3].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[3]};
  bind core[4].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[4]};
  bind core[5].cluster_local_switch_wr_if
               {cluster_local_link_rd_channel[5]};
```

```
  bind core[6].cluster_local_switch_wr_if
                {cluster_local_link_rd_channel[6]};
  bind core[7].cluster_local_switch_wr_if
                {cluster_local_link_rd_channel[7]};
  bind edram_0_mem.ocn_wr_if {cluster_local_link_rd_channel[8]};
  bind edram_1_mem.ocn_wr_if {cluster_local_link_rd_channel[9]};
  bind edram_2_mem.ocn_wr_if
{cluster_local_link_rd_channel[10]};
  bind edram_3_mem.ocn_wr_if
{cluster_local_link_rd_channel[11]};
  bind cluster_scheduler.cluster_local_switch_wr_if
                {cluster_local_link_rd_channel[12]};
  Queue cluster_local_link_wr_channel[13];
  bind cluster_local_switch.ocn_wr_if {
      cluster_local_link_wr_channel[0],
      cluster_local_link_wr_channel[1],
      cluster_local_link_wr_channel[2],
      cluster_local_link_wr_channel[3],
      cluster_local_link_wr_channel[4],
      cluster_local_link_wr_channel[5],
      cluster_local_link_wr_channel[6],
      cluster_local_link_wr_channel[7],
      cluster_local_link_wr_channel[8],
      cluster_local_link_wr_channel[9],
      cluster_local_link_wr_channel[10],
      cluster_local_link_wr_channel[11],
      cluster_local_link_wr_channel[12],
      ocn_wr_if};

  bind core[0].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[0]};
  bind core[1].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[1]};
  bind core[2].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[2]};
  bind core[3].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[3]};
  bind core[4].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[4]};
  bind core[5].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[5]};
  bind core[6].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[6]};
  bind core[7].cluster_local_switch_rd_if
                {cluster_local_link_wr_channel[7]};
```

```
  bind edram_0_mem.ocn_rd_if
{cluster_local_link_wr_channel[8]};
  bind edram_1_mem.ocn_rd_if
{cluster_local_link_wr_channel[9]};
  bind edram_2_mem.ocn_rd_if
{cluster_local_link_wr_channel[10]};
  bind edram_3_mem.ocn_rd_if
{cluster_local_link_wr_channel[11]};

  bind cluster_scheduler.cluster_local_switch_rd_if
               {cluster_local_link_wr_channel[12]};
};

PE ReorderController {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
};

PE TrafficManager("TrafficManagerConfig.cfg") {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if;
  Queue queue_0, queue_1, queue_2, queue_3, queue_4,
  Queue queue_5, queue_6, queue_7;
};

PE Deparser {
  QueueRdI ocn_rd_if;
  QueueWrI ocn_wr_if, egress;
};

PE NPU("NPUConfig.cfg") implements ControlPlaneAgentS {
  QueueRdI ingress;
  QueueWrI egress;

  Splitter splitter;
  Parser parser("Parser.cfg");
  Scheduler scheduler("Scheduler.cfg");
  Cluster cluster("Cluster.cfg")[8];
  ReorderController roc;
  TrafficManager tm("TrafficManager.cfg");
  Deparser deparser;
  ControlPlaneAgent cpagent;
  Mem mct_0_mem("OffChipConfig.cfg");
  Mem edram_payload_mem("memory.cfg");
  MemoryManager memory_manager;
  Router ocn_00, ocn_10, ocn_20;
  Router ocn_01, ocn_11, ocn_21;
```

```
  Router ocn_02, ocn_12, ocn_22;

  // --- ocn_00 ---
  Queue ocn00_rd_channel[4];
  Queue ocn00_wr_channel[4];

  //ocn_00 //"deparser", "edram_payload_mem", "ocn_10", "ocn_01"
  bind ocn_00.ocn_rd_if              {ocn00_rd_channel[0],
ocn00_rd_channel[1], ocn00_rd_channel[2], ocn00_rd_channel[3]};
  bind ocn_00.ocn_wr_if              {ocn00_wr_channel[0],
ocn00_wr_channel[1], ocn00_wr_channel[2], ocn00_wr_channel[3]};

  bind deparser.ocn_rd_if            {ocn00_wr_channel[0]};
  bind deparser.ocn_wr_if            {ocn00_rd_channel[0]};

  bind edram_payload_mem.ocn_rd_if   {ocn00_wr_channel[1]};
  bind edram_payload_mem.ocn_wr_if   {ocn00_rd_channel[1]};

  // --- ocn_10 ---
  Queue ocn10_rd_channel[4];
  Queue ocn10_wr_channel[4];

  //ocn_10 //"ocn_00", "tm", "roc", "ocn_20", "ocn_11"
  bind ocn_10.ocn_rd_if              {ocn00_wr_channel[2],
ocn10_rd_channel[0], ocn10_rd_channel[1], ocn10_rd_channel[2],
ocn10_rd_channel[3]};
  bind ocn_10.ocn_wr_if              {ocn00_rd_channel[2],
ocn10_wr_channel[0], ocn10_wr_channel[1], ocn10_wr_channel[2],
ocn10_wr_channel[3]};

  bind tm.ocn_rd_if                  {ocn10_wr_channel[0]};
  bind tm.ocn_wr_if                  {ocn10_rd_channel[0]};

  bind roc.ocn_rd_if                 {ocn10_wr_channel[1]};
  bind roc.ocn_wr_if                 {ocn10_rd_channel[1]};

  // --- ocn_20 ---
  Queue ocn20_rd_channel[3];
  Queue ocn20_wr_channel[3];

  //ocn_20 //"ocn_10", "cluster[6]", "cluster[4]", "ocn_21"
  bind ocn_20.ocn_rd_if
{ocn10_wr_channel[2],ocn20_rd_channel[0],ocn20_rd_channel[1],ocn
20_rd_channel[2]};
  bind ocn_20.ocn_wr_if
{ocn10_rd_channel[2],ocn20_wr_channel[0],ocn20_wr_channel[1],ocn
20_wr_channel[2]};
```

```
  bind cluster[6].ocn_wr_if          {ocn20_rd_channel[0]};
  bind cluster[6].ocn_rd_if          {ocn20_wr_channel[0]};

  bind cluster[4].ocn_wr_if          {ocn20_rd_channel[1]};
  bind cluster[4].ocn_rd_if          {ocn20_wr_channel[1]};

  // --- ocn_01 ---
  Queue ocn01_rd_channel[4];
  Queue ocn01_wr_channel[4];

  //ocn_01 //"ocn_00", "parser", "splitter", "ocn_11", "ocn_02"
  bind ocn_01.ocn_rd_if
{ocn00_wr_channel[3],ocn01_rd_channel[0],ocn01_rd_channel[1],ocn
01_rd_channel[2],ocn01_rd_channel[3]};
  bind ocn_01.ocn_wr_if
{ocn00_rd_channel[3],ocn01_wr_channel[0],ocn01_wr_channel[1],ocn
01_wr_channel[2],ocn01_wr_channel[3]};

  bind parser.ocn_wr_if              {ocn01_rd_channel[0]};
  bind parser.ocn_rd_if              {ocn01_wr_channel[0]};

  bind splitter.ocn_wr_if            {ocn01_rd_channel[1]};
  bind splitter.ocn_rd_if            {ocn01_wr_channel[1]};

  // --- ocn_11 ---
  Queue ocn11_rd_channel[3];
  Queue ocn11_wr_channel[3];

  //ocn_11 //"ocn_01", "ocn_10", "scheduler", "ocn_21", "ocn_12"
  bind ocn_11.ocn_rd_if
{ocn01_wr_channel[2],ocn10_wr_channel[3],ocn11_rd_channel[0],ocn
11_rd_channel[1],ocn11_rd_channel[2]};
  bind ocn_11.ocn_wr_if
{ocn01_rd_channel[2],ocn10_rd_channel[3],ocn11_wr_channel[0],ocn
11_wr_channel[1],ocn11_wr_channel[2]};

  bind scheduler.ocn_wr_if           {ocn11_rd_channel[0]};
  bind scheduler.ocn_rd_if           {ocn11_wr_channel[0]};

  // --- ocn_21 ---
  Queue ocn21_rd_channel[3];
  Queue ocn21_wr_channel[3];

  //ocn_21 //"ocn_11", "ocn_20", "cluster[0]", "cluster[1]",
"ocn_22"
```

```
  bind ocn_21.ocn_rd_if
{ocn11_wr_channel[1],ocn20_wr_channel[2],ocn21_rd_channel[0],ocn
21_rd_channel[1],ocn21_rd_channel[2]};
  bind ocn_21.ocn_wr_if
{ocn11_rd_channel[1],ocn20_rd_channel[2],ocn21_wr_channel[0],ocn
21_wr_channel[1],ocn21_wr_channel[2]};

  bind cluster[0].ocn_wr_if          {ocn21_rd_channel[0]};
  bind cluster[0].ocn_rd_if          {ocn21_wr_channel[0]};

  bind cluster[1].ocn_wr_if          {ocn21_rd_channel[1]};
  bind cluster[1].ocn_rd_if          {ocn21_wr_channel[1]};

  // --- ocn_02 ---
  Queue ocn02_rd_channel[4];
  Queue ocn02_wr_channel[4];

  //ocn_02 //"ocn_01", "cpagent", "memory_manager", "mct_0_mem",
"ocn_12"
  bind ocn_02.ocn_rd_if
{ocn01_wr_channel[3],ocn02_rd_channel[0],ocn02_rd_channel[1],ocn
02_rd_channel[2],ocn02_rd_channel[3]};
  bind ocn_02.ocn_wr_if
{ocn01_rd_channel[3],ocn02_wr_channel[0],ocn02_wr_channel[1],ocn
02_wr_channel[2],ocn02_wr_channel[3]};

  bind cpagent.ocn_wr_if             {ocn02_rd_channel[0]};
  bind cpagent.ocn_rd_if             {ocn02_wr_channel[0]};

  bind memory_manager.wr_if          {ocn02_rd_channel[1]};
  bind memory_manager.rd_if          {ocn02_wr_channel[1]};

  bind mct_0_mem.ocn_wr_if           {ocn02_rd_channel[2]};
  bind mct_0_mem.ocn_rd_if           {ocn02_wr_channel[2]};

  //  --- ocn_12 ---
  Queue ocn12_rd_channel[3];
  Queue ocn12_wr_channel[3];
  //ocn_12  //"ocn_02", "ocn_11", "cluster[2]", "cluster[3]",
"ocn_22"
  bind ocn_12.ocn_rd_if
{ocn02_wr_channel[3],ocn11_wr_channel[2],ocn12_rd_channel[0],ocn
12_rd_channel[1],ocn12_rd_channel[2]};
  bind ocn_12.ocn_wr_if
{ocn02_rd_channel[3],ocn11_rd_channel[2],ocn12_wr_channel[0],ocn
12_wr_channel[1],ocn12_wr_channel[2]};
```

```
  bind cluster[2].ocn_wr_if          {ocn12_rd_channel[0]};
  bind cluster[2].ocn_rd_if          {ocn12_wr_channel[0]};

  bind cluster[3].ocn_wr_if          {ocn12_rd_channel[1]};
  bind cluster[3].ocn_rd_if          {ocn12_wr_channel[1]};

  //  --- ocn_22 ---
  Queue ocn22_rd_channel[2];
  Queue ocn22_wr_channel[2];
  //ocn_22  //"ocn_12", "ocn_21", "cluster[5]", "cluster[7]"
  bind ocn_22.ocn_rd_if
{ocn12_wr_channel[2],ocn21_wr_channel[2],ocn22_rd_channel[0],ocn
22_rd_channel[1]};
  bind ocn_22.ocn_wr_if
{ocn12_rd_channel[2],ocn21_rd_channel[2],ocn22_wr_channel[0],ocn
22_wr_channel[1]};

  bind cluster[5].ocn_wr_if          {ocn22_rd_channel[0]};
  bind cluster[5].ocn_rd_if          {ocn22_wr_channel[0]};

  bind cluster[7].ocn_wr_if          {ocn22_rd_channel[1]};
  bind cluster[7].ocn_rd_if          {ocn22_wr_channel[1]};

  bind splitter.ingress              {ingress};
  bind deparser.egress               {egress};

};
PE PacketGenerator("PacketGeneratorConfig.cfg") {
  QueueWrI out;
};
PE SortedLogger {
  QueueRdI in;
  QueueWrI out;
};
PE Logger("LoggerConfig.cfg") {
  QueueRdI in;
  QueueWrI out;
};
PE PacketSink {
  QueueRdI in;
};
PE top("TopConfig.cfg") {
  Queue IKI, IKE;
  PacketGenerator packet_generator;
  NPU npu("NPU.cfg");
  ControlPlane control_plane;
  bind packet_generator.out {IKI};
```

```
  bind npu.ingress {IKI};
  bind npu.egress {IKE};
  bind control_plane.cpa {npu};
  Queue logger_out;
  PacketSink sink;

  Logger logger;
  bind logger.in {IKE};
  bind logger.out {logger_out};

  bind sink.in    {logger_out};

};
```

Chapter 10: Appendix – B

RMT FAD

```
import tcam;

interface QueueRdI, QueueWrI, MemI;
service ControlPlaneAgentS;

CE Queue("QueueConfig.cfg") implements QueueRdI, QueueWrI;
CE Memory implements MemI;

service MemoryManagerS;

PE MemoryManager implements MemoryManagerS {
  MemI mem_port[];
};

PE Multiplexer {
  QueueRdI mux_input[];
  QueueWrI mux_output[];
};

PE Demultiplexer {
  QueueRdI demux_input[];
  QueueWrI demux_output[];
};

PE PacketGenerator("PacketGeneratorConfig.cfg") {
  QueueWrI out;
};

PE Logger("LoggerConfig.cfg") {
  QueueRdI in;
};

PE Parser {
  QueueRdI parser_in;
  QueueWrI parser_out;

  MemoryManagerS memory_manager;
};

PE Deparser {
```

```
  QueueRdI deparser_in;
  QueueWrI deparser_out;

  MemoryManagerS memory_manager;
};

PE Selector {
  QueueRdI select_in;
  QueueWrI select_out;
};

PE MatchTable {
  QueueRdI table_in;
  QueueWrI table_out;
  TcamSearchEngineS tse_port;
};

PE VLIWAction {
  QueueRdI action_in;
  QueueWrI action_out;
};

PE MatchStage {
  QueueRdI match_stage_in;
  QueueWrI match_stage_out;

  Queue selector_to_match, match_to_action;

  Selector selector;
  MatchTable match_table;
  VLIWAction vliw_action;
  TcamSearchEngine tse;

  bind selector.select_in {match_stage_in};
  bind selector.select_out {selector_to_match};
  bind match_table.table_in {selector_to_match};
  bind match_table.table_out {match_to_action};
  bind vliw_action.action_in {match_to_action};
  bind vliw_action.action_out {match_stage_out};
  bind match_table.tse_port {tse};
};

PE Pipeline {
  QueueRdI pipe_in;
  QueueWrI pipe_out;

  Queue to_stage1;
```

```
Queue from_stage32;
Queue stage_to_stage[31];
Queue demux_to_parser[16];
Queue parser_to_mux[16];

Parser parser[16];
Demultiplexer pre_parse_demux;
Multiplexer post_parse_mux;
MatchStage match_stage[32];
Deparser deparser;

MemoryManagerS memory_manager;

bind pre_parse_demux.demux_input[0] {pipe_in};

bind pre_parse_demux.demux_output[0] {demux_to_parser[0]};
bind pre_parse_demux.demux_output[1] {demux_to_parser[1]};
bind pre_parse_demux.demux_output[2] {demux_to_parser[2]};
bind pre_parse_demux.demux_output[3] {demux_to_parser[3]};
bind pre_parse_demux.demux_output[4] {demux_to_parser[4]};
bind pre_parse_demux.demux_output[5] {demux_to_parser[5]};
bind pre_parse_demux.demux_output[6] {demux_to_parser[6]};
bind pre_parse_demux.demux_output[7] {demux_to_parser[7]};
bind pre_parse_demux.demux_output[8] {demux_to_parser[8]};
bind pre_parse_demux.demux_output[9] {demux_to_parser[9]};
bind pre_parse_demux.demux_output[10] {demux_to_parser[10]};
bind pre_parse_demux.demux_output[11] {demux_to_parser[11]};
bind pre_parse_demux.demux_output[12] {demux_to_parser[12]};
bind pre_parse_demux.demux_output[13] {demux_to_parser[13]};
bind pre_parse_demux.demux_output[14] {demux_to_parser[14]};
bind pre_parse_demux.demux_output[15] {demux_to_parser[15]};

bind parser[0].parser_in {demux_to_parser[0]};
bind parser[1].parser_in {demux_to_parser[1]};
bind parser[2].parser_in {demux_to_parser[2]};
bind parser[3].parser_in {demux_to_parser[3]};
bind parser[4].parser_in {demux_to_parser[4]};
bind parser[5].parser_in {demux_to_parser[5]};
bind parser[6].parser_in {demux_to_parser[6]};
bind parser[7].parser_in {demux_to_parser[7]};
bind parser[8].parser_in {demux_to_parser[8]};
bind parser[9].parser_in {demux_to_parser[9]};
bind parser[10].parser_in {demux_to_parser[10]};
bind parser[11].parser_in {demux_to_parser[11]};
bind parser[12].parser_in {demux_to_parser[12]};
bind parser[13].parser_in {demux_to_parser[13]};
bind parser[14].parser_in {demux_to_parser[14]};
```

```
bind parser[15].parser_in {demux_to_parser[15]};

bind parser[0].parser_out {parser_to_mux[0]};
bind parser[1].parser_out {parser_to_mux[1]};
bind parser[2].parser_out {parser_to_mux[2]};
bind parser[3].parser_out {parser_to_mux[3]};
bind parser[4].parser_out {parser_to_mux[4]};
bind parser[5].parser_out {parser_to_mux[5]};
bind parser[6].parser_out {parser_to_mux[6]};
bind parser[7].parser_out {parser_to_mux[7]};
bind parser[8].parser_out {parser_to_mux[8]};
bind parser[9].parser_out {parser_to_mux[9]};
bind parser[10].parser_out {parser_to_mux[10]};
bind parser[11].parser_out {parser_to_mux[11]};
bind parser[12].parser_out {parser_to_mux[12]};
bind parser[13].parser_out {parser_to_mux[13]};
bind parser[14].parser_out {parser_to_mux[14]};
bind parser[15].parser_out {parser_to_mux[15]};

bind parser[0].memory_manager {memory_manager};
bind parser[1].memory_manager {memory_manager};
bind parser[2].memory_manager {memory_manager};
bind parser[3].memory_manager {memory_manager};
bind parser[4].memory_manager {memory_manager};
bind parser[5].memory_manager {memory_manager};
bind parser[6].memory_manager {memory_manager};
bind parser[7].memory_manager {memory_manager};
bind parser[8].memory_manager {memory_manager};
bind parser[9].memory_manager {memory_manager};
bind parser[10].memory_manager {memory_manager};
bind parser[11].memory_manager {memory_manager};
bind parser[12].memory_manager {memory_manager};
bind parser[13].memory_manager {memory_manager};
bind parser[14].memory_manager {memory_manager};
bind parser[15].memory_manager {memory_manager};

bind post_parse_mux.mux_input[0] {parser_to_mux[0]};
bind post_parse_mux.mux_input[1] {parser_to_mux[1]};
bind post_parse_mux.mux_input[2] {parser_to_mux[2]};
bind post_parse_mux.mux_input[3] {parser_to_mux[3]};
bind post_parse_mux.mux_input[4] {parser_to_mux[4]};
bind post_parse_mux.mux_input[5] {parser_to_mux[5]};
bind post_parse_mux.mux_input[6] {parser_to_mux[6]};
bind post_parse_mux.mux_input[7] {parser_to_mux[7]};
bind post_parse_mux.mux_input[8] {parser_to_mux[8]};
bind post_parse_mux.mux_input[9] {parser_to_mux[9]};
bind post_parse_mux.mux_input[10] {parser_to_mux[10]};
```

```
bind post_parse_mux.mux_input[11] {parser_to_mux[11]};
bind post_parse_mux.mux_input[12] {parser_to_mux[12]};
bind post_parse_mux.mux_input[13] {parser_to_mux[13]};
bind post_parse_mux.mux_input[14] {parser_to_mux[14]};
bind post_parse_mux.mux_input[15] {parser_to_mux[15]};

bind post_parse_mux.mux_output[0] {to_stage1};

bind match_stage[0].match_stage_in {to_stage1};
bind match_stage[1].match_stage_in {stage_to_stage[0]};
bind match_stage[2].match_stage_in {stage_to_stage[1]};
bind match_stage[3].match_stage_in {stage_to_stage[2]};
bind match_stage[4].match_stage_in {stage_to_stage[3]};
bind match_stage[5].match_stage_in {stage_to_stage[4]};
bind match_stage[6].match_stage_in {stage_to_stage[5]};
bind match_stage[7].match_stage_in {stage_to_stage[6]};
bind match_stage[8].match_stage_in {stage_to_stage[7]};
bind match_stage[9].match_stage_in {stage_to_stage[8]};
bind match_stage[10].match_stage_in {stage_to_stage[9]};
bind match_stage[11].match_stage_in {stage_to_stage[10]};
bind match_stage[12].match_stage_in {stage_to_stage[11]};
bind match_stage[13].match_stage_in {stage_to_stage[12]};
bind match_stage[14].match_stage_in {stage_to_stage[13]};
bind match_stage[15].match_stage_in {stage_to_stage[14]};
bind match_stage[16].match_stage_in {stage_to_stage[15]};
bind match_stage[17].match_stage_in {stage_to_stage[16]};
bind match_stage[18].match_stage_in {stage_to_stage[17]};
bind match_stage[19].match_stage_in {stage_to_stage[18]};
bind match_stage[20].match_stage_in {stage_to_stage[19]};
bind match_stage[21].match_stage_in {stage_to_stage[20]};
bind match_stage[22].match_stage_in {stage_to_stage[21]};
bind match_stage[23].match_stage_in {stage_to_stage[22]};
bind match_stage[24].match_stage_in {stage_to_stage[23]};
bind match_stage[25].match_stage_in {stage_to_stage[24]};
bind match_stage[26].match_stage_in {stage_to_stage[25]};
bind match_stage[27].match_stage_in {stage_to_stage[26]};
bind match_stage[28].match_stage_in {stage_to_stage[27]};
bind match_stage[29].match_stage_in {stage_to_stage[28]};
bind match_stage[30].match_stage_in {stage_to_stage[29]};
bind match_stage[31].match_stage_in {stage_to_stage[30]};

bind match_stage[0].match_stage_out {stage_to_stage[0]};
bind match_stage[1].match_stage_out {stage_to_stage[1]};
bind match_stage[2].match_stage_out {stage_to_stage[2]};
bind match_stage[3].match_stage_out {stage_to_stage[3]};
bind match_stage[4].match_stage_out {stage_to_stage[4]};
bind match_stage[5].match_stage_out {stage_to_stage[5]};
```

```
  bind match_stage[6].match_stage_out {stage_to_stage[6]};
  bind match_stage[7].match_stage_out {stage_to_stage[7]};
  bind match_stage[8].match_stage_out {stage_to_stage[8]};
  bind match_stage[9].match_stage_out {stage_to_stage[9]};
  bind match_stage[10].match_stage_out {stage_to_stage[10]};
  bind match_stage[11].match_stage_out {stage_to_stage[11]};
  bind match_stage[12].match_stage_out {stage_to_stage[12]};
  bind match_stage[13].match_stage_out {stage_to_stage[13]};
  bind match_stage[14].match_stage_out {stage_to_stage[14]};
  bind match_stage[15].match_stage_out {stage_to_stage[15]};
  bind match_stage[16].match_stage_out {stage_to_stage[16]};
  bind match_stage[17].match_stage_out {stage_to_stage[17]};
  bind match_stage[18].match_stage_out {stage_to_stage[18]};
  bind match_stage[19].match_stage_out {stage_to_stage[19]};
  bind match_stage[20].match_stage_out {stage_to_stage[20]};
  bind match_stage[21].match_stage_out {stage_to_stage[21]};
  bind match_stage[22].match_stage_out {stage_to_stage[22]};
  bind match_stage[23].match_stage_out {stage_to_stage[23]};
  bind match_stage[24].match_stage_out {stage_to_stage[24]};
  bind match_stage[25].match_stage_out {stage_to_stage[25]};
  bind match_stage[26].match_stage_out {stage_to_stage[26]};
  bind match_stage[27].match_stage_out {stage_to_stage[27]};
  bind match_stage[28].match_stage_out {stage_to_stage[28]};
  bind match_stage[29].match_stage_out {stage_to_stage[29]};
  bind match_stage[30].match_stage_out {stage_to_stage[30]};
  bind match_stage[31].match_stage_out {from_stage32};

  bind deparser.deparser_in {from_stage32};

  bind deparser.deparser_out {pipe_out};

  bind deparser.memory_manager {memory_manager};
};

PE Buffer {
  QueueRdI buffer_in;
  QueueWrI buffer_out;
};

PE ControlPlaneAgent implements ControlPlaneAgentS {
  QueueRdI from_egress;
  QueueWrI to_ingress;
};

PE ControlPlane {
  ControlPlaneAgentS cpa;
};
```

```
PE IngressMultiplexer {
  QueueRdI packet_in;
  QueueRdI from_agent;
  QueueWrI output;
};

PE EgressDemultiplexer {
  QueueRdI input;
  QueueWrI packet_out;
  QueueWrI to_agent;
};

PE RMT implements ControlPlaneAgentS {
  QueueRdI rmt_in;
  QueueWrI rmt_out;

  Queue to_buffer, from_buffer, agent_to_ingress,
egress_to_agent, to_ingress, from_egress;
  Pipeline ingress_pipeline, egress_pipeline;
  Buffer buffer;
  ControlPlaneAgent cp_agent;
  IngressMultiplexer ingress_mux;
  EgressDemultiplexer egress_demux;
  MemoryManager memory_manager;
  Memory mem;

  bind ingress_mux.packet_in {rmt_in};
  bind ingress_mux.from_agent {agent_to_ingress};
  bind ingress_mux.output {to_ingress};

  bind egress_demux.input {from_egress};
  bind egress_demux.packet_out {rmt_out};
  bind egress_demux.to_agent {egress_to_agent};

  bind ingress_pipeline.pipe_in {to_ingress};
  bind ingress_pipeline.pipe_out {to_buffer};
  bind ingress_pipeline.memory_manager {memory_manager};

  bind buffer.buffer_in {to_buffer};
  bind buffer.buffer_out {from_buffer};

  bind egress_pipeline.pipe_in {from_buffer};
  bind egress_pipeline.pipe_out {from_egress};
  bind egress_pipeline.memory_manager {memory_manager};

  bind cp_agent.to_ingress {agent_to_ingress};
```

```
  bind cp_agent.from_egress {egress_to_agent};

  bind memory_manager.mem_port[0] {mem};
};

PE top {
  Queue ingress_queue, egress_queue;
  PacketGenerator pktgen;
  Logger logger;
  RMT rmt;
  ControlPlane control_plane;

  bind pktgen.out {ingress_queue};
  bind rmt.rmt_in {ingress_queue};
  bind rmt.rmt_out {egress_queue};
  bind logger.in {egress_queue};
  bind control_plane.cpa {rmt};
};
```

TCAM FAD

```
interface TcamMemI;
service TcamS, TcamSearchEngineS;

CE TcamMemory implements TcamMemI;

PE Tcam("TcamConfig.cfg") implements TcamS {};

PE DefaultTcamController implements TcamSearchEngineS{
  TcamS tcam_port;
  TcamMemI mem;
};

PE TcamSearchEngine implements TcamSearchEngineS {
  Tcam main_tcam;
  DefaultTcamController controller;
  TcamMemory TLMMemory;

  bind controller.tcam_port {main_tcam};
  bind controller.mem {TLMMemory};
};
```