TOWARDS AN IDEAL EXECUTION ENVIRONMENT FOR
PROGRAMMABLE NETWORK SWITCHES

A Thesis
Presented to
The Graduate Faculty of The University of Akron

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Michael Geoffrey Gruesen
August, 2016
TOWARDS AN IDEAL EXECUTION ENVIRONMENT FOR PROGRAMMABLE NETWORK SWITCHES

Michael Geoffrey Gruesen

Thesis

Approved: 

Advisor
Dr. Andrew Sutton

Faculty Reader
Dr. Timothy O’Neil

Faculty Reader
Dr. Zhong-Hui Duan

Accepted:

Dean of the College
Dr. John Green

Dean of the Graduate School
Dr. Chand Midha

Date

Department Chair
Dr. David Steer
ABSTRACT

Software Defined Networking (SDN) aims to create more powerful, intelligent networks that are managed using programmed switching devices. Applications for these SDN switches should be target independent, while being efficiently translated to the platform's native machine code. However network switch vendors do not conform to any standard, and contain different capabilities and features that vary between manufacturers.

The Freeflow Virtual Machine (FFVM) is a modular, fully programmable virtual switch that can host compiled network applications. Applications are compiled to native object libraries and dynamically loaded at run time. The FFVM provides the necessary data and computing resources required by applications to process packets. This work details the many implementation approaches investigated and evaluated in order to define a suitable execution environment for hosted network applications.
ACKNOWLEDGEMENTS

First, I would like to thank Dr. Andrew Sutton for being my advisor and giving me the opportunity to work as your research assistant during my graduate studies. I am truly grateful for all that I have learned while working with you.

I would also like to thank my wife and family for their support while pursuing this degree.

A special thanks goes out to Hoang Nguyen, who developed the Steve programming language and compiler along side the Freeflow project to provide network applications for hosting.

To the members of Flowgrammable, namely Jasson Casey, Dr. Paul Gratz, Dr. Alex Sprintson, and Luke McHale; Thank you for being a great research team to work with.

This material is based upon work supported by the National Science Foundation under Grant No. 1423322.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>viii</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
</tbody>
</table>

## CHAPTER

### I. INTRODUCTION

1.1 Background .................................. 2
1.2 Goals ........................................ 5
1.3 Contributions ................................. 5
1.4 Organization ................................ 6

### II. RELATED WORK

2.1 Software Defined Networking ............ 7
2.2 Languages .................................... 10
2.3 Heterogeneous Computing .................. 12
2.4 Event Processing Frameworks ............. 15

### III. EARLY INVESTIGATION: HARDWARE ABSTRACTION

3.1 DPDK .......................................... 18
3.2 Netmap ...................................... 20
3.3 ODP .......................................... 22
3.4 Conclusions ................................. 24
### IV. EARLY INVESTIGATION: PROCESSING INSTRUCTIONS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 RISC-V</td>
<td>25</td>
</tr>
<tr>
<td>4.2 HSAIL</td>
<td>27</td>
</tr>
<tr>
<td>4.3 Conclusions</td>
<td>28</td>
</tr>
</tbody>
</table>

### V. FREEFLOW

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Application Hosting</td>
<td>30</td>
</tr>
<tr>
<td>5.2 Packet Context</td>
<td>33</td>
</tr>
<tr>
<td>5.3 Virtual Machine</td>
<td>36</td>
</tr>
<tr>
<td>5.4 Ports</td>
<td>37</td>
</tr>
<tr>
<td>5.5 Tables</td>
<td>40</td>
</tr>
<tr>
<td>5.6 Instructions</td>
<td>41</td>
</tr>
<tr>
<td>5.7 Memory</td>
<td>42</td>
</tr>
<tr>
<td>5.8 Threading</td>
<td>43</td>
</tr>
<tr>
<td>5.9 ABI</td>
<td>45</td>
</tr>
<tr>
<td>5.10 Conclusion</td>
<td>46</td>
</tr>
</tbody>
</table>

### VI. EXPERIMENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 L1 Receive - Endpoint</td>
<td>48</td>
</tr>
<tr>
<td>6.2 L2 Forwarding - Wire</td>
<td>49</td>
</tr>
<tr>
<td>6.3 Threading Models</td>
<td>53</td>
</tr>
<tr>
<td>6.4 Results</td>
<td>55</td>
</tr>
<tr>
<td>6.5 Discussion</td>
<td>58</td>
</tr>
</tbody>
</table>
VII. CONCLUSIONS .............................................. 60

7.1 Future Work .............................................. 61

BIBLIOGRAPHY .................................................. 62
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>A flow table that maps input ports to output ports.</td>
<td>41</td>
</tr>
<tr>
<td>6.1</td>
<td>Freeflow Endpoint and Flowcap throughput and bandwidth performance.</td>
<td>57</td>
</tr>
<tr>
<td>6.2</td>
<td>Freeflow wire driver performance metrics, including relative speedup.</td>
<td>58</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Execution environment approaches.</td>
<td>1</td>
</tr>
<tr>
<td>3.1</td>
<td>DPDK L2 forwarding driver.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>DPDK’s port abstraction on the spectrum.</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>Netmap driver functionality.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Netmap’s port abstraction on the spectrum.</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>ODP framework.</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>ODP’s port abstraction on the spectrum.</td>
<td>23</td>
</tr>
<tr>
<td>4.1</td>
<td>RISC-V execution environment on the spectrum.</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>HSAIL execution environment on the spectrum.</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>FFVM provides computational and memory resources to applications.</td>
<td>29</td>
</tr>
<tr>
<td>5.2</td>
<td>Freeflow application logic spans across multiple boundaries.</td>
<td>30</td>
</tr>
<tr>
<td>5.3</td>
<td>Freeflow applications define a list of decoders and tables.</td>
<td>31</td>
</tr>
<tr>
<td>5.4</td>
<td>An IPv4 Ethernet frame.</td>
<td>34</td>
</tr>
<tr>
<td>5.5</td>
<td>An example binding environment for an IPv4 Ethernet frame.</td>
<td>36</td>
</tr>
<tr>
<td>5.6</td>
<td>The Freeflow virtual switch architecture.</td>
<td>38</td>
</tr>
<tr>
<td>5.7</td>
<td>Freeflow port UML diagram.</td>
<td>39</td>
</tr>
<tr>
<td>5.8</td>
<td>Freeflow packet context memory models.</td>
<td>43</td>
</tr>
</tbody>
</table>
6.1 The Freeflow STA wire state machine. ........................................ 54
6.2 The Freeflow TPP wire state machine. ........................................ 56
CHAPTER I

INTRODUCTION

This thesis presents a series of experiments searching to define an ideal execution environment for Software Defined Networking (SDN) applications. By exploring the different implementation methods utilized in the domain of SDN, namely pure software and hardware, the benefits and deficits of each can be evaluated and contribute to a concrete specification. In a pure software implementation, most of the underlying hardware components have been abstracted, providing more flexibility and compatibility at the cost of performance. A pure hardware implementation will translate a high level language into optimized native instructions to keep performance high, but narrows the set of languages supported. In Figure 1.1, the spectrum for these two methods is shown.

![Figure 1.1: Execution environment approaches.](image)

To translate high level code to a target device’s native instruction set, more work has to be done by the compiler to support both the language and the device.
Knowledge of the target’s architecture, including specialized hardware accelerators, must be well defined in order optimize execution. However this approach is not always ideal, especially in the domain of SDN where network switch architecture varies greatly between vendors in terms of capabilities and features. This contributes to a lack of generalization in modern networking where end-users are typically only able to specify configuration settings. In order to bridge the gap, hardware components can be abstracted to provide low level access to resources available in the system. A suitable execution environment would lie somewhere between these two, taking advantage of powerful compilers that can lower high level code while also leveraging low level resource interfaces raised from the hardware.

1.1 Background

Networking architecture has become the focus of many facets of computing with an increased reliance on network data. As more and more devices become connected and users produce and consume data at an ever-increasing rate, the way we accommodate these needs in networking infrastructure has prompted the need for change. In order to service the needs of users, network engineers require more functionality from their equipment than basic switching and routing. Their networks need to be more intelligent, and allow for more powerful user generated network applications. Unfortunately conventional network switching devices are fairly static and rigid, and do not provide the means to network engineers to create custom applications that suit their particular needs.
A network switch is composed of two high level components, a control plane and a forwarding, or data, plane. The control plane manages the configuration and state of the device, as well as compiled application tenancy; whereas the data plane is responsible for executing forwarding behavior over network traffic flows in the system. These components are tightly coupled which hinders how a network application might be able to abstract their functionality. At the upper level, control plane management is, more or less, a fairly open entity. Users can configure their networks and applications running on them within the confines of the interface exposed by that particular switch vendor. To apply changes made in the control plane, which may alter the configuration of the data plane, usually the device must be rebooted. Most changes made to resources used by the data plane and applications can only occur when the system is starting up. This has much to do with the fact that the applications running on these devices are created by the vendors themselves, who understand the underlying architecture in the system and are able to push the application logic down into the hardware. The result is high performance networking applications that utilize hardware accelerators and are tuned for that particular system. However this comes at a great cost, flexibility. Network engineers are at the mercy of networking device manufacturers when they want to mold their network to suit their needs. The data plane in a network switch remains a black box, with features and capabilities varying greatly between vendors. This lack of transparency makes it difficult to model an abstract machine that can be targeted by networking applications.
Research in the domain of SDN investigates the decoupling of the two planes, giving users the ability to create networking applications that can respond to changes in network traffic flows in real time. The emphasis in this model is to allow a single control plane to manage numerous data planes in one or many switches as a unified entity. Most of the work in this field revolves primarily around virtual network switches, building on concepts provided by the OpenFlow specification [1]. We chose to focus on the implementation of a single programmable network device, where the control and data plane exist within the same system. This allowed us to produce a specification for an abstract machine for networking applications. The machine defines memory and object models, program execution semantics, a set of required operations, access to guaranteed resources, and the types of objects that it operates on and their behaviors. It must also be able to support a variety of high level programming languages as well as platforms and network processor architectures. Optimization support for specialized instruction execution and offloading computation to hardware accelerators and/or co-processors is also a key requirement to keep performance on par. Languages are able to interact with the virtual machine through an application binary interface (ABI) that defines symbols and rules that dictate how they can utilize the runtime system. This approach has produced an instance of an abstract machine that provides the necessary resources and capabilities required to support network application execution.
1.2 Goals

It is clear that there is need for runtime support for compiled networking applications. The virtual machine needed to provide functionality for both parts of a network switch, the control and data planes, and allow applications to have control and access to these resources at the user level. In addition to servicing the needs of applications, the runtime needs to be supported on a variety of architectures and utilize the materialized and abstracted resources available. Freeflow aims to fill the gap between high level network programming languages and low level switch architectures for the purpose of SDN.

1.3 Contributions

The contributions presented in this thesis come from numerous experiments with low level networking hardware. These experiments helped characterize the problem domain with respect to low level abstraction of network interfaces and high level language translation to a native instruction set architecture (ISA). The culmination of lessons learned are found in the Freeflow virtual machine (FFVM) implementation, the main focus in this work. The experiments evaluated include:

- DPDK and Netmap - Low level port abstractions
- RISC-V - Native instruction execution
- Threading - Execution models
1.4 Organization

The thesis is organized as follows. In Chapters 3 and 4, the early investigations into hardware abstraction and processing instructions are discussed, respectively. This section journals the different approaches that were considered in the design and implementation throughout this project, and explains the benefits, deficits, and lessons learned from each approach. Chapter 5 covers the design and implementation of the Freeflow system, including application hosting, the virtual machine, the runtime support library and the application binary interface. This chapter discusses the details of the contributions made towards this work at length. Chapter 6 presents the experiments conducted as well as some initial evaluation metrics collected. These demonstrate basic networking device functionality for emulating Ethernet port endpoints and cross-connects over TCP sockets, and provide a baseline which can be used to further optimize the system. In the final chapter, Chapter 7, the conclusions drawn from the research conducted are discussed. Future work with respect to lessons learned throughout the development of this project and integration with other SDN programming languages and frameworks are also covered.
CHAPTER II

RELATED WORK

Though there are many approaches to solving the numerous problems SDN brings about, there is a lack of consensus on what the “correct” way is. This field of research continues to grow, and overlap with other research interests is becoming more common. The major sections in this chapter discuss the current SDN solutions, heterogeneous computing, and event processing frameworks.

2.1 Software Defined Networking

SDN provides a networking device model where the control and data planes are decoupled from one another; a single control plane can be responsible for distributing application execution across a set of network switches viewed as a unified data plane instance. Applications generally operate at the control plane level and rely on distributed platforms to push the logic down to the hardware level. Allowing users to define their own control and data plane logic enables the creation and adoption of custom networking protocols. The amount of control gained with respect to the shape and behavior of the network being managed creates more intelligent networks.
2.1.1 OpenFlow

The OpenFlow [1] SDN model defines a switch that separates and distributes the control and data planes. Centralized controllers aggregate data plane switches and communicate over a dedicated channel utilizing the OpenFlow protocol. The protocol defines messages for packet I/O, data plane resource modifications, as well as state and configuration queries.

An OpenFlow switch’s data plane is can be viewed as a flow table that defines forwarding behavior for packets that match a particular set of fields within a packet header. Packets that do not match entries in a flow table are sent to the controller for further inspection. The controller is able to update flow tables by installing new forwarding rules for packets that match ones sent from the data plane.

The one downside to the OpenFlow model is the bottleneck created when utilizing a messaging channel between two different physical devices. Freeflow focuses on the implementation of a virtual switch where the control and data plane logic are contained within the same device.

2.1.2 DPDK

Intel’s Data Plane Development Kit (DPDK) [2] provides a framework that allows programmers to create highly optimized data plane applications. This platform utilizes custom drivers that provide raw access to networking devices and computational resources. The system is tuned to natively support Intel brand hardware and allows users to create C applications that execute in a runtime environment.
Freeflow’s port abstraction resides at a slightly higher level than DPDK to support more networking interfaces, and as such DPDK ports have been integrated into the FFVM.

2.1.3 Netmap

The Netmap project [3] is comprised of three functional components contained in a single kernel module. With the Netmap kernel module, user space applications have access to network interface cards, the host operating system networking stack, as well as VALE virtual ports and Netmap “pipes”. VALE is the Netmap implementation of a software Ethernet controller that can instantiate virtual ports accessible through the Netmap API.

The broad class of network interfaces that Netmap supports is desirable in the FFVM implementation. To compensate for the lack of control provided in the abstraction, Netmap ports have been implemented in the Freeflow port abstraction.

2.1.4 OpenDataPlane

The OpenDataPlane (ODP) project [4] establishes an open source API for defining networking data plane applications. ODP provides application portability over numerous networking architectures. Use of hardware accelerators can be achieved without burdening the user with a required knowledge of the capabilities and features present in an execution environment.

ODP and the FFVM lie at roughly the same level with respect to high level language support and low level hardware abstractions. FFVM focuses more on flex-
ibility and programmability with respect to data plane processing, whereas ODP focuses on exposing APIs to interface with other SDN execution environments.

2.1.5 Open Virtual Switch

Open Virtual Switch (OVS) [5] enables the distribution of data plane applications over an aggregation of switching devices. OVS replaces the bridge between hyper-visors running on top of a distributed machine that is typically provided by OS kernels. The OVS controller is capable of managing pure software and/or hardware switches, and supports the usage of specialized hardware accelerators.

Freeflow applications can take advantage of the distribution of data plane switch logic by way of the FFVM. Integration with the OVS distributed architecture would mitigate northbound and southbound communication from the hardware to the application, and vice versa.

2.2 Languages

SDN programming languages exemplify the features and capabilities programmers want to have supported in an SDN application execution environment. These give indications of what a suitable execution environment should provide in order to support a large number of higher level languages. The following two sections describe a couple of the more mature SDN programming languages referenced.
2.2.1 P4

P4 [6] provides a network programming language for protocol independent packet parsers. These parsers are akin to the decoding stages executed in Freeflow packet processing pipelines, but executed on dedicated hardware. The forwarding model for P4 uses match+action tables (similar to decision tables) to match on “parsed representations” (PR) generated by the parsers. A PR is the data structure which holds the parsing state, which gives access to protocol header fields.

Given the popularity of the P4 programming language, support is being considered in future implementations of the FFVM. Packet parsing units can be application-specific integrated circuits (ASICs), that are highly optimized for packet decoding operations. Utilizing the P4 language could take advantage of these types of processors available to an FFVM execution environment.

2.2.2 POF

Protocol Oblivious Forwarding (POF) [7] defines a small ISA that can be executed on POF SDN switches. The POF model represents a packet processing pipeline as a series of matching tables that perform packet header decoding when invoked. The decoded header fields are stored in meta data as an offset and length pair, referred to as “search keys”.

The pipeline composition and storage strategies are similar to the FFVM application pipeline and context binding environment, respectively. Integrating POF has been a consideration for the project to broaden the set of supported languages.
2.3 Heterogeneous Computing

Much of the research in the domain of SDN in regards to runtime environments encompass many of the problems found in heterogeneous computing platforms. These issues tend to revolve around the utilization of optimized hardware computational devices. This relates to the needs of networking programming where a data plane is expected to be able to offload certain computation to hardware components, such as encryption, checksums, and even packet header processors. Below is a listing of the relevant projects in this domain that were studied during the design and implementation of the Freeflow system.

2.3.1 HSA

HSA [8] provides a specification for creating a heterogeneous system architecture that supports the execution of applications across a variety of computational resources. Interfacing with these different computing devices requires knowledge of each device’s memory model and execution semantics. In order to program against them the system must provide a uniform view that each device conforms to. Many of the core components defined in the HSA specification were considered during the implementation of the FFVM. Unified views of memory and time as well as concurrent thread communication attempt to follow this model to orchestrate multiple computational devices or computing cores.
2.3.2 CUDA

NVIDIA’s Compute Unified Device Architecture (CUDA) [9] for General Purpose Graphics Computing Units (GPGPUs) allows programmers to utilize a graphics card to execute applications on a SIMD device. Each graphics card hosts a different set of features and capabilities, such as the number of cores available, memory architecture, and instruction sets. The CUDA library enables developers to write a single instance of a program and deploy it on a wide selection of supported GPUs. This is achieved by utilizing the NVIDIA CUDA Compiler (NVCC) which translates the higher level source code to the device’s native machine code by way of an intermediate representation (IR). The NVIDIA Virtual Machine (NVVM) IR is based on the Low Level Virtual Machine (LLVM) IR used by the Clang compiler. Once the code has been lowered to the NVVM IR it can be translated to the parallel thread execution virtual machine and ISA, PTX. These instructions are executed natively by any CUDA enabled GPU device.

The narrowed view of CUDA with respect to heterogeneous computing is similar to the FFVM design in that specialized hardware components that posses different capabilities, features, and instruction sets need to be considered during compilation.

2.3.3 Open Compute Language

Khronos Group’s Open Compute Language (OpenCL) [10] offers an open, cross-platform standard for programming parallel devices. Supported devices are personal computers, servers, and systems-on-a-chip (SOCs). The Standard Portable
IR (SPIR), based on LLVM IR, was evolved into a similarly open and cross-platform language called SPIR-V. This IR utilizes LLVM as the target device, and allows for device specific code to be generated from a single application source.

OpenCL focuses on the more general problem discussed in the previous topic, supporting a larger class of computational devices. FFVM is similar to OpenCL in attempting to solve the general problem of managing and optimizing the usage of different computing resources on a single system.

2.3.4 NetASM

The customized networking assembly language NetASM [11] defines an instruction set architecture (ISA) that is tuned specifically for networking programming. Instructions for modifying packet headers and state as well as table and specialized operations are natively supported. In addition to the networking oriented instructions the set also contains basic logical, arithmetic, and control-flow operations. Applications written in NetASM can model the execution environment as either a traditional Turing complete register machine or an extended abstract machine developed by the authors.

The NetASM ISA approaches the problem of processing instructions natively in an execution environment at the machine level. We had considered this approach, as explained in Chapter 4, but abandoned it as re-implementing basic arithmetic, logical, and control flow operations to be interpreted at run time would negatively
impact performance. In essence, the solution implements the MIPS ISA with some additional specialized network programming instructions.

2.3.5 RISC V

RISC-V [12] specifies an extensible ISA, which encompasses the standard arithmetic, logical and control-flow operations with room for a number of user-defined instructions. Support for variable length instructions, multiple address spaces (32, 64, 128 bit), and modern parallelization architectures are also provided. RISC-V instruction execution requires the usage of RISC-V CPUs, currently provided through an ISA simulator (Spike), or a VM running a modified Linux kernel.

An extensible ISA that provides basic ALU operations as a core instruction set that is added to is a more fitting choice for implementing a network programming ISA. However, lack of support for executing RISC-V instructions in commercial processors prevents the approach from being embraced. FFVM instructions need to be executed on physical hardware in a native machine code, which is not currently possible with this ISA.

2.4 Event Processing Frameworks

One of the goals in SDN is to make networks more reactive, and intelligent. Event processing frameworks provide many of the mechanisms required to implement typical networking application behavior. The following sections describe two processing
frameworks considered, but the project has not reached the state of maturity where either can be integrated.

2.4.1 Seastar
The Seastar project [13] is an open source C++ framework that optimizes applications for modern architectural features supported in many high performance, highly parallel servers. These features include the use of different networking stacks, modern language features for concurrent programming, and low overhead lock-free message passing between computing cores. The combination of these features lets the framework host powerful event driven applications. Support for Seastar warrants high end hardware that can make full use of the framework and was deemed too exotic to integrate at this time.

Seastar’s high performance execution is enabled by the use of many computational and memory resources, rarely found in commercial hardware. In its current state, the FFVM is not yet mature enough to utilize this exotic event processing framework. Enterprise servers possess an abundance of processors and memory, and would be required to host Seastar applications. During consideration, compilation of the Seastar framework was unsuccessful due to the lack of resources available in the testing hardware.

2.4.2 Libevent
Libevent [14] exposes an API for executing function callbacks triggered by file descriptor, signal, and timeout events to simplify the development of event driven network
server applications. Supporting numerous polling mechanisms on different platforms (Linux, Mac, Windows) allows applications to be highly portable and efficient.

Integration of the libevent framework was considered but ruled out due to the dependency on file descriptors. For logical based ports (sockets) this is not an issue. However in the case of other network interfaces that do not rely on file descriptors, accommodation becomes difficult. The ability to create delegate functions to handle event call backs would be a great addition to Freeflow. A mechanism to bridge the gap between the dependency of the framework and the Freeflow port abstraction goals could be developed in the future to use libevent.
CHAPTER III
EARLY INVESTIGATION: HARDWARE ABSTRACTION

The initial research for this thesis into the domain of SDN was based on the utilization of frameworks that provide low level access to networking resources. In order to maintain high performance and control in user level application space, these devices need to be abstracted. Network interfaces, both logical and physical, can expose a variety of capabilities and features that need to be properly encapsulated. This chapter discusses the evaluation of low level port abstractions provided by DPDK (Section 3.1) and Netmap (Section 3.2) mid-level port abstractions implemented in ODP (Section 3.3) as well as the conclusions drawn in Section 3.4.

3.1 DPDK

Intel’s DPDK provides users with a framework on which data plane applications can be built utilizing highly optimized constructs and device drivers. At the time of this investigation, the project was in the early stages of development (version 1.0, 1.1). The drivers provided by the development kit are written specifically for Intel brand networking devices. Systems that lack compatible hardware can provision a virtual machine to take advantage of the optimized port drivers available. In order to execute an application with the DPDK runtime environment, the host operating system
requires some additional modifications as well. A custom networking kernel module allows applications to bypass the host systems native networking stack, reducing the number of copies made between user and kernel space. Memory page size is also increased to reduce the number of page faults.

(a) Default L2 forwarding behavior.
(b) Modified L2 forwarding behavior.

Figure 3.1: DPDK L2 forwarding driver.

The goal of the experiment is to create a virtual “wire” between two Ethernet ports, such that data received by port A is sent by port B, and vice versa. An example application, L2 Forward, provided in the development kit served as a starting point for this experiment. The example creates an external wire between pairs of ports, where port A sends data to port B, and B sends to A. In order to create an internal wire, the driver for each port needed to be changed. Figure 3.1 illustrates the behavior
models used in the experiment. The modified implementation has each port reading from its counterpart’s receive queue and sending a copy from itself. The destination address fields have to be re-factored from being statically hard coded in the port driver to a dynamic, mutable property.

Execution of the modified application gives inconsistent results, and can require system restarts between runs to reconfigure the virtual machine when errors occur. This is attributed to performance and abstraction penalties suffered from the use of the virtual machine as the runtime environment host. Though DPDK’s optimized port drivers deliver solid performance, the results show that utilizing this framework for abstracting low level port resources would narrow the field of compatible physical devices to target.

![Diagram](image)

Figure 3.2: DPDK’s port abstraction on the spectrum.

3.2 Netmap

The second framework evaluated for port abstraction comes from the Netmap project. Netmap provides low level access to network interfaces by diverting the flow of data to and from the interface away from the operating system, and towards a custom
device. Receive and transmit queues are mapped into user space where the raw data can be operated on directly rather than through the operating system network stack. Figure 3.3 illustrates the mechanism provided by Netmap.

(a) User space applications use the kernel to access resources.
(b) Netmap gives direct access to hardware memory resources.

Figure 3.3: Netmap driver functionality.

The portability factor makes Netmap a great candidate for a low level port abstraction framework, supporting hardware NIC, logical (e.g. UDP, TCP, UNIX sockets), and virtual (provided by the Netmap VALE software switch) ports. Moving the memory address space for a network device into user space eliminates the need for coherency resolution through the operating system kernel, and gives an increase in performance while processing raw packet data. However there is a lack of control
over the networking interfaces to allow applications to monitor the state and alter the configuration of ports. Netmap ports are currently supported by the Freeflow port abstraction to overcome the missing functionality.

![Netmap Ports](image)

Figure 3.4: Netmap’s port abstraction on the spectrum.

3.3 ODP

The OpenDataPlane project provides an API for programming network data plane applications across multiple platforms. Currently, software implementations exist for Linux and DPDK back ends as references for design and integration purposes. ODP allows for different back ends in order to support a broader class of networking devices. Figure 3.5 shows the possible architectural models that can be constructed using ODP.

ODP’s aim to provide flexibility and performance is mirrored in the design of the FFVM, and support for the framework is being investigated. The C API exposed gives enough user space control over networking interfaces while minimizing the performance cost incurred by hardware abstraction. Integration of ODP ports
(a) ODP and Linux APIs expose the underlying hardware to applications. (b) ODP and DPDK provide accelerated driver support to applications.

Figure 3.5: ODP framework.

into the FFVM port abstraction is currently being evaluated. Figure 3.6 shows where ODP’s port implementation falls in the virtualization-materialization spectrum.

Figure 3.6: ODP’s port abstraction on the spectrum.
3.4 Conclusions

During these three investigations, the design and implementation of the FFVM port abstraction evolved. The proper level of abstraction lies just above the middle of the virtualization-materialization spectrum, where programmers can maintain a high level of control over networking interfaces while utilizing optimized operational mechanisms provided in the target system.
CHAPTER IV
EARLY INVESTIGATION: PROCESSING INSTRUCTIONS

After analyzing and experimenting with the hardware abstraction problems in network switch devices, the focus shifted to the other end of the spectrum: translating high level instructions. Not every hardware component in a system needs to be abstracted up to the user level, but they need to be properly utilized. In this chapter, two approaches to solving the translation problem are considered. The first uses the extensible ISA named RISC-V in Section 4.1, and the second evaluates the usage of the HSA intermediate representation, HSAIL [15], in Section 4.2. In Section 4.3, the conclusions from each path of processing instructions from high level languages are evaluated.

4.1 RISC-V

RISC-V provides a base ISA that leaves room for the addition of custom, user defined instructions that can be executed natively on a CPU architecture that implements the ISA. For an abstract network switch, this ISA is well suited to be natively supported, as typical network application operations can be optimized. Specialized instructions can be added for packet decoding, table matching, and packet modification (e.g. push/pop header tags). Many languages support these mechanisms from a high
level perspective, but lack support to translate them down to native instructions. Network processors vary between vendors, but typically do not implement a full, general purpose ISA that would be present on a CPU. For example, floating point arithmetic is not used in network processing and as a result, NPUs do not contain floating point units. The hardware is fine tuned for the domain in which it operates, and the native machine ISA needs to reflect that.

The core of any RISC-V implementation is a base integer ISA that can have many optional instruction set “extensions” added to it. Outside of the core, each extension can be considered standard or non-standard. Standard extensions provide semantics for multiplication/division, atomic, and floating point operations and do not cause conflicts with one another. Non-standard extensions are unmanaged. They can be highly specialized and/or optimized but do not provide compatibility guarantees with standard extensions. The modular nature of the ISA allows for the full customization of a RISC-V variant to suit the needs of a particular implementation.

As the RISC-V ISA is only able to be natively executed by virtual CPUs, an interpreter is required to process the instruction set. This approach would allow for the execution of natively specialized network instructions but sacrifices performance for the added flexibility. The FFVM instruction set needs to be flexible to support specialized networking operations but must also execute efficiently. Figure 4.1 shows where the RISC-V instruction set lies on the virtualization-materialization spectrum.
4.2 HSAIL

The HSA intermediate language, or HSAIL, is the intermediate representation which serves as an abstract native instruction set for HSA compatible parallel computing devices. These devices include multi-core CPU’s, GPU’s, and other hardware accelerators. HSAIL programs can be executed natively on devices that are HSA compliant, or can be just-in-time compiled to the target’s machine code and loaded if not.

As the HSA focuses on the general problem of abstracting device functionality in the domain of parallel computing, networking processors and the specialized hardware accelerators contained in most switches would fall into the non-compliant category at this time. However the strategy implemented in HSAIL, and many other heterogeneous computing platforms, is becoming the commonplace approach to enable single source programming for multiple target devices with varying capabilities and features. Pushing the application logic into the hardware is the most efficient way to process instructions. Figure 4.2 plots the HSAIL instruction set on the spectrum.
4.3 Conclusions

The emphasis on the balance between flexibility and efficiency with respect to processing instructions is growing in many facets of computing. As hardware evolves and becomes more disjoint from the CPU, the ability to efficiently utilize the individual components lessens. The modular ISA provided by RISC-V presents a way to build on a simple core instruction set that can be supported by multiple devices in a system and allow for a single source application to execute natively. However, the lack of physical RISC-V processors forces less efficient execution environments to be used. In order to achieve optimal usage of multiple computing units driven by a single application, the instructions executed on a given unit must be translated to the target device’s native ISA.
CHAPTER V
FREEFLOW

The Freeflow project aims to bridge the gap between high level network programming languages and low level networking hardware by defining an ideal execution environment for network applications. Evaluating the benefits and deficits with respect to flexibility and performance provides a more accurate level of translation or abstraction to use. In Figure 5.1 the Freeflow system architecture is illustrated.

![Freeflow System Architecture Diagram](image)

Figure 5.1: FFVM provides computational and memory resources to applications.

This chapter elaborates on the Freeflow programmable virtual switch implementation details. The first few sections describe how applications are hosted and how packet data is represented and manipulated. In the remainder of the chapter, the virtual machine and its components are discussed. These include object models for
ports and tables, dynamic instruction execution, as well as the memory and threading models supported.

5.1 Application Hosting

Freeflow applications provide the logic for the control and data planes in the switch. This is a departure from common SDN solutions, where the two planes are treated as separate entities and typically distributed across many devices. The reason for blurring the line between the two parts is to reduce the overhead penalty that is incurred in the former model. By allowing the application to straddle the line between the control and data planes, the logic it provides is able to be pushed into the appropriate hardware and executed efficiently.

![Figure 5.2: Freeflow application logic spans across multiple boundaries.](image)

Networking applications operate on packets, utilizing information found inside of nested protocol headers to determine the appropriate action to take. In Freeflow, applications and the virtual machine operate on packet contexts that store...
contextual information about a packet in the system. Applications use these contexts to process packets in a series of stages that extract protocol information and apply rule matching logic to them. A typical application pipeline is illustrated in Figure 5.3.

![Figure 5.3: Freeflow applications define a list of decoders and tables.](image)

Freeflow applications are loaded dynamically through application binary files, compiled to native shared object libraries. Currently only one instance of an application binary can be loaded into a FFVM process space. The FFVM calls functions, exported as symbols from the binary, to manipulate the state of the application. Freeflow application state is defined as:

- **init** - All exported symbol handles have been resolved.

- **ready** - The application is loaded and able to start.

- **running** - The application is being executed.

- **stopped** - The application is halted.
The functions that control the application life cycle all require a handle to the host data plane in order to provide dynamic allocation of resources. These functions are:

\[
\begin{align*}
\text{ff\_load}(dp) ; \\
\text{ff\_unload}(dp) ; \\
\text{ff\_start}(dp) ; \\
\text{ff\_stop}(dp) ; \\
\text{ff\_process} (cxt) ;
\end{align*}
\]

The \text{ff\_load} function initializes global data plane resources required by the application, such as tables. When successful the application state is set to “ready”, or is left in an “init” state when errors occur. During FFVM tear down, a call to \text{ff\_unload} will release the allocated resources if the application is not in a “running” state.

Applications “learn” about state and configuration changes to the data plane they are hosted in through events. The \text{ff\_start} function sets the application state to “running”, starting packet processing and allowing events to be handled. A call to start is only valid when the state of the application is “ready” or “stopped”. The \text{ff\_stop} function halts packet processing, disables event handlers, and sets the state to “stopped”.

32
The application defines the forwarding behavior for a data plane by providing a packet processing pipeline. After a packet is received and exits ingress processing, the FFVM calls the process function on a packet context. In order to forward packets, the application must provide the definition of a packet processing pipeline for the FFVM to execute.

5.2 Packet Context

Network packets are arranged as nested protocol headers, which contain fields that describe the structure and contents of a particular layer. Since the Freeflow data plane has no knowledge of any protocol structures (i.e. it is protocol independent), it operates on contextual information extracted by applications and stored in a context object. The meta data contained in a context allows the data plane to provide robust network functionality and execute a variety of network applications.

5.2.1 Packets

In networking, packets represent raw data that have been transmitted over some media with protocol headers for each layer contained in the packet. Each header gives information about the structure and state of the current protocol being processed. Figure 5.4 shows an example Ethernet frame (packet) containing an IP (v4) header.

The layout for each protocol is far from uniform, and fields within headers do not always align to the traditional byte-aligned boundaries (e.g. Ethernet MAC addresses which have a width of 48 bits). Storing these values in memory results in
wasted space, as larger data structures would need to be utilized. Structure packing pragmas and bit width specifiers can be used to overlap memory regions when the desired bit width is less than the standard width for that type (e.g. `std::int_64:48`). However storing a copy of the packet data would result in coherency issues, negatively impacting performance. Instead the fields within headers are stored in a binding environment, discussed in the following section.

5.2.2 Context

The packet context provides contextual information about an associated packet in the system. A context is where applications store input, control, and decoder information, as well as application meta data. This is also where they build action lists. Input
information consists of data about the packets arrival into the system, such as the input logical and physical ports. The control information maintains the control flow of a context as it traverses processing pipelines.

```c
struct Decode_info {
    uint16_t pos; // Packet decoding offset
    Environment headers; // Saved packet header locations
    Environment fields; // Saved packet fields locations
};

struct Context {
    Ingress_info input;
    Control_info ctrl;
    Decode_info decode;
    Packet packet; // A handle to packet memory
    Metadata metadata; // Application-defined metadata
    Action_list actions; // A sequence of instructions
};
```

As application packet decoders execute, they store the position, or offset, and length of the desired fields as pairs in a binding environment, referred to as the decode information. This environment notes the offset of a field within the current protocol header, as well as the offset for each protocol header within the packet buffer. Utilizing this heavy weight approach to referencing protocol field information enforces a more precise data extraction plan and eliminates coherency issues. Figure 5.5 shows the resulting binding environment created after decoding the Ethernet source and destination fields, as well as the IPv4 protocol and destination fields.

The raw packet data is accessed through a handle held by the context. Depending on the capabilities of the port that received the packet, this handle may point
to a Freeflow packet buffer or dedicated port memory. Meta data provides “scratch” space for applications to store additional information during pipeline processing. The action list is composed of instructions that are to be applied to a packet after exiting an application pipeline but before egress processing. These instructions are used to modify fields within a packet, and are explained further in Section 5.6.

5.3 Virtual Machine

The Freeflow Virtual Machine is composed of modular parts that a programmer can assemble into a virtual switch. This flexibility allows for the instantiation of numerous switches with varying features and capabilities. In general, switches require control and data planes as well as compiled network applications to drive each component. A FFVM virtual switch can be assembled by creating a data plane instance, in which port resources are added and an application is loaded. User space drivers provision
the machine with these components and execute compiled applications. An example driver is shown in the following listing.

```cpp
// Create a data plane instance named 'dp1'.
ff::dataplane dp = "dp1";

// Add virtual (internal) ports and two TCP sockets.
dp.add_virtual_ports();
ff::Port* port1 = new ff::Port_tcp(1);
ff::Port* port2 = new ff::Port_tcp(2);
dp.add_port(port1);
dp.add_port(port2);

// Load a sample application named 'sample.app'.
dp.load_application("sample.app");

// Set data plane configuration to 'up',
// starting execution.
dp.up();
```

The driver makes a virtual switch with a single data plane, named “dp1”, that contains the predefined virtual ports and two TCP ports, with i.d.s 1 and 2 respectively. After allocating port resources, an application named “sample.app” is loaded, allowing the system to perform table configuration and import the pipeline processing symbols. Figure 5.6 illustrates the orchestration of the components utilized in the example driver listing.

5.4 Ports

Ports act as the main source of I/O for network applications. They provide the means to receive and send packets that are entering and leaving the system. As an
abstract type, the port interface is fairly simple. Any derived port object needs only implement the following four functions:

- **Send** - Transmit packet data.
- **Receive** - Retrieve packet data.
- **Up** - Puts the port into a usable state, data can be sent and received.
- **Down** - Disables port functionality.

Port objects can be classified as being physical, logical, or virtual. A physical port represents a hardware networking interface, e.g. Ethernet cards. Logical ports represent software networking constructs that utilize file descriptors to act as an endpoint for communication. An illustration of the port object UML can be found...
Virtual ports are derived from logical ports, and provide specialized functionality for the system. To further classify the port type, they can be either seen as an input port, where data from an external source can be ingressed into the system, or an output port, which sends data to other external or internal devices. Currently input ports can be either logical or physical, whereas output ports can be any port type (i.e. logical, physical, virtual).

![Port UML diagram]

When packets are received by a port object, the memory that holds the raw packet data and the context are allocated during the ingress phase. The underlying data store for the packet and context are owned and managed by the port itself.
Memory can be allocated from the ports internal resources (i.e. memory mapped from a physical device address space) if present, or by the FFVM’s global buffer pool. This gives a greater amount of flexibility in that the packet and context can exist in same memory region, potentially improving locality. As packets leave the system during the egress phase, this memory is released back to the appropriate devices. Details about memory management can be found in Section 5.7.

After initializing a context with a new packet, the port passes the context to the application pipeline for processing. When the context exits the application pipeline, the resulting action list is applied to the packet. Finally the context enters the egress stage where the packet is forwarded to the designated output port set in the context, or dropped.

5.5 Tables

In network switches, tables are used to match properties of packets with user-defined forwarding behaviors. They can be defined using a variety of data structures and algorithms that implement them, but generally are categorized as exact, prefix, or wild card. All three table types are currently implemented by the FFVM, though support for prefix and wild card is incomplete.

Each entry in a flow table contains a key that is compared to a certain field within the packet. Key types vary between match table types. For an exact match table the keys are integers of a set width, defined by the application. These keys are used to aggregate traffic containing similar characteristics into flows. Flows can be
viewed as programs, or functions, that get executed when a packet matches their key.

Each flow defines a set of operations, or instructions, that are applied to packets. Table 5.1 depicts a simple flow table that creates a virtual “wire” between two ports, where traffic received by one port is sent out the other.

<table>
<thead>
<tr>
<th>Input Port</th>
<th>Flow Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>output(2);</td>
</tr>
<tr>
<td>2</td>
<td>output(1);</td>
</tr>
<tr>
<td>miss</td>
<td>drop;</td>
</tr>
</tbody>
</table>

Table 5.1: A flow table that maps input ports to output ports.

5.6 Instructions

The FFVM hosts network applications that are dynamically loaded and executed at run time. To avoid using an interpreter, applications must be compiled to native shared objects. By translating the high level language down to the target machine code, the execution of instructions provided in the binaries is more efficient. These instructions are encapsulated in functions that can be executed on a variety of computational devices. This allows the FFVM to take advantage of hardware accelerators tuned for network processing.
5.7 Memory

Memory for raw packet data and contexts is allocated by port objects during the ingress processing phase. The FFVM is optimized to allow for zero-copy packet processing, where raw packet data resides in a physical device’s dedicated memory. In certain cases device packet buffers contain extra memory for user data, which can store the context object to improve locality. Another benefit of storing the packet and context in the same region of memory is that the two entities can grow dynamically; the context grows as actions are appended to it’s action list and the packet grows when new protocol headers are pushed onto a header binding environment. When a port object does not possess the capability to provide persistent packet memory, a global packet context buffer can be allocated by the system. Figure 5.8 illustrates the two memory models currently supported by the FFVM.

A global buffer consists of a packet buffer, a context, and a non-zero i.d. Packet buffers are pre-allocated 2048 byte regions that can easily accommodate IEEE 802.3 Ethernet frames that have a maximum size of 1523 bytes. The packet buffers are over sized to allow for the insertion of new protocol headers that may result during or after pipeline processing. Contexts residing in a global buffer contain a reference to the packet buffer, which by default points to a FFVM packet buffer. The i.d. gives the offset into the global buffer pool, and allows ports to return the i.d. back to the pool’s free list. FFVM’s free list is implemented as a min-heap, providing the next available packet buffer that can be allocated. During creation, and upon
Freeflow allocates context memory that references packet data in port memory.

Port memory can accommodate dynamically sized packet and context objects.

Figure 5.8: Freeflow packet context memory models.

de-allocation, a packet context buffer is set to a null state; each field in the context is default initialized and the packet size is set to 0. In this state, referencing a context or the packet buffer it is associated with is undefined.

5.8 Threading

The FFVM supports multiple threading architectures to maximize execution resource utilization. Behavior for FFVM components, such as port objects and application pipelines, can be defined as free functions in a driver and executed in its own thread. Threads contribute to the modular nature of the FFVM and can be treated as build-
ing blocks to produce different execution models. FFVM provides a simple threading interface built on the POSIX thread library (pthread) [16], that executes a preallocated work routine and passes the thread i.d. as the argument. Synchronization mechanisms and configuration settings, or attributes, are optionally provided. The following listing shows how a FFVM thread can be constructed with an initial i.d. and work routine, and also re-assigned while the thread is not in a “running” state.

```c
void* port_work(void* arg) {
    // ... 
}

void* port_work2(void* arg) {
    // ... 
}

int main(int argc, char** argv) {
    ff::Thread thread = { 1, port_work }; 
    thread.run(); 
    // ... 
    thread.halt(); 
    thread.assign(2, port_work2); 
    thread.run(); 
    // ... 
    thread.halt(); 
    return 0; 
}
```

In addition to the threading interface, the FFVM also provides shared data structures such as queues and object pools. Concurrent queue implementations exist for locked and lock-free access, allowing for more flexibility in the threading architecture being modeled. Object pools allocate reusable global resources that can also be shared across thread boundaries.
The FFVM application binary interface exposes a set of symbols and calling conventions that can be utilized by applications at run time. These “system calls” allow hosted applications to request system resources. In this section the more pertinent functions are discussed.

```
ff_create_table(dp, width, n, type);
ff_delete_table(dp, tbl);
```

A call to the `ff_create_table` function returns a newly allocated flow table in the given data plane instance `dp` with an application defined key `width` in bytes, an initial size of `n` entries, and matching table `type`. The `ff_delete_table` call releases the resources allocated in the given table `tbl`.

```
ff_lookup_flow(tbl, k);
ff_insert_flow(tbl, k, f);
ff_remove_flow(tbl, k);
```

Flow entry matching is executed with a call to `ff_lookup_flow` which searches for a key `k` in a flow table `tbl`. Table modifications can be made using `ff_insert_flow` to add a new key `k` with the flow `f` to table `tbl`, and `ff_remove_flow` to remove the key `k` from the table `tbl`. If the key already exists when the call to insert is made, the associated flow object is updated, whereas a call to remove a non-existent key from a table has no effect.
\begin{verbatim}
ff_output(cxt, p);
ff_drop(cxt);
\end{verbatim}

The `ff_output` function sets the `output_port` field in a context `cxt` to the id of the given port `p`. A call to the `ff_drop` function effectively sets the `output_port` field in the context `cxt` to the data plane drop port id, but terminates the further processing in the pipeline stage it was invoked from.

5.10 Conclusion

Freeflow embodies the contributions made in this thesis. All of the components developed make Freeflow a framework for creating virtual programmable switches. The modular approach allows the VM to be re-configurable at run time where it can dynamically change shape with respect to resources, ports and memory, and functionality from real time updates to forwarding rules. We choose this model in order to create and evaluate different switch architectures and threading models. In order to work towards an ideal execution environment for hosted network applications, Freeflow gives the ability to test and narrow the set of requirements that are necessary to achieve this goal.
CHAPTER VI
EXPERIMENTS

The experiments conducted to evaluate the functionality and performance of the FFVM emulate network switch behavior using multiple threading architectures. To drive the FFVM, a high level network application language called Steve [17, 18] provides the hosted applications. Each experiment is comprised of a FFVM driver, which creates an instance of a virtual switch, and the compiled network application. Input for the experiments, network traffic, is generated by an external application named Flowcap that transmits the contents of a packet capture (PCAP) file. PCAP files contain live network data that has been formatted for use by 3rd party libraries, such as libpcap [19]. Flowcap can also be used to send and receive the contents of a PCAP file, which serves as a baseline measure for one of the experiments. The retransmission of the capture file simulates a steady flow of input for each experiment to evaluate its performance. The traffic is “framed” over TCP sockets to emulate raw Ethernet frames that would be received at the lowest network protocol layer. A small protocol header occupies the first 4 bytes of each frame to denote the length of the proceeding data. In the following sections, the goals for each experiment are explained along with the different threading architectures implemented, and lastly the performance metrics collected during each trial are discussed.
6.1 L1 Receive - Endpoint

The baseline test for the FFVM is an endpoint application, where a simple server is constructed that accepts a single connection and reports the receive rate. This test evaluates the maximum rate that data can be received and processed by an application. Below is a simplified listing of the driver that creates an instance of a virtual switch and executes the “endpoint” application.

```cpp
// Build a server socket that will accept network connections.
Ipv4_socket_address addr(Ipv4_address::any(), 5000);
Ipv4_stream_socket server(addr);

// Pre-create all standard ports.
Port_eth_tcp port1(1);

// Configure the dataplanes ports before loading applications.
ff::Dataplane dp ="dp1";
dp.add_port(&port1);
dp.load_application("apps/endpoint.app");
dp.up();

while (running) {
    poll(server, port1);

    if (server.can_read())
        accept_new_connection(server);

    if (port1.can_read()) {
        ff::Context ctx;
        port1.receive(ctx);

        dp.get_application() -> process(ctx);

        if (ctx.has_output_port())
            ctx.output_port() -> send(ctx);
    }
}
6.2 L2 Forwarding - Wire

The second experiment conducted creates an L2 wire that behaves similarly to the modified DPDK example discussed in Chapter 3. Setup for this experiment extends the “endpoint” driver by adding an additional TCP Ethernet port, and utilizes slightly more functional application named “wire.app”. During pipeline processing, each context’s output port is set to the “opposite” of the input port. The wire driver listing is shown below.

```cpp
// Build a server socket that will accept network connections.
Ipv4_socket_address addr(Ipv4_address::any(), 5000);
Ipv4_stream_socket server(addr);

// Pre-create all standard ports.
Port_eth_tcp port1(1);
Port_eth_tcp port2(2);

// Configure the dataplane ports before loading applications.
ff::Dataplane dp = "dp1";
dp.add_port(&port1);
dp.add_port(&port2);
dp.load_application("apps/wire.app");
dp.up();

while (running) {
    poll(server, port1, port2);

    if (server.can_read())
        accept_new_connection(server);

    if (port1.can_read()) {
```
Different threading models were considered in the wire implementation, and shed light on optimizations that could be made to the shared resources provided in the FFVM. Further explanations of the different threading models used are found in Section 6.3. In a multi-threaded architecture the drivers for ports are defined as free functions, where each port receives packets and after pipeline processing places a copy of the context into a transmit queue tied to each port. A simplified listing of the initial multi-threaded wire driver is shown below.

```c
// Global resources.
ff::Context cxt;
port1.receive(cxt);

dp.get_application() -> process(cxt);
if (cxt.has_output_port())
  cxt.output_port() -> send(cxt);

if (port2.can_read()) {
  ff::Context cxt;
  port2.receive(cxt);

  dp.get_application() -> process(cxt);

  if (cxt.has_output_port())
    cxt.output_port() -> send(cxt);
}
```
while (ports[id].up()){
    ff::Context ctx;
    ports[id].receive(ctx);
    dp.get_application()->process(ctx);
    if (int out_id = ctx.output_port())
        send_queue[out_id].push(ctx);
    if (ff::Context out_ctx = send_queue[id].pop()) {
        ports[id].send(out_ctx);
    }
} return ;

main(){
    // Build a server socket that will accept network
    // connections.
    Ipv4_socket_address addr(Ipv4_address::any(), 5000);
    Ipv4_stream_socket server(addr);
    // Configure the dataplanes ports before loading
    // applications.
    dp.add_port(&ports[0]);
    dp.add_port(&ports[1]);
    dp.load_application("apps/wire.app");
    dp.up();

    while (running) {
        poll(server);
        if (server.can_read())
            accept_new_connection_and_launch_thread(server);
    }
    return 0;
}

In this setup the only resource that can cause concurrency issues is the global send queue associated with each port. The initial implementation of the FFVM concurrent queue utilized mutex locks to control read/write access from multiple threads. However the bottleneck created by the locking mechanism drastically reduced perfor-
formance. To provide a concurrent lock-free queue, an implementation was integrated from the Boost C++ libraries [20]. Performance improved greatly with the use of lock-free queues, yet there were still issues with memory usage. Each port creates a local context and pushes a copy into the send queue for the output port, and context is then copied again before being sent from the designated output port. This warranted the need for a reusable object pool, where contexts and packet data would reside in a pre-allocated system buffer. Rather than copying the context to and from the send queues, the buffer i.d. is enqueued. Ports are able to access the context and packet through the buffer pool using the buffer i.d. as the offset into the pool’s underlying data store. The final optimization added in this experiment was to have ports cache a local store of processed buffer i.d.s and copy the contents of the local store when it is “full”. Performance increased greatly over the previous driver implementation by utilizing re-usable packet buffers and the local store of buffer i.d.s. A listing for the final wire driver is show below.

```cpp
// Global resources.
ff::Port_eth_tcp ports[2] = { 1, 2 };
ff::Object_pool<ff::Buffer> buffer_pool;
ff::Queue_concurrent<int> send_queue[2];

port_thread_work(){
    int id = get_thread_id();

    // Create local stores for processed buffer ids.
    int local_size_max = 1024;
    std::vector<int> local_cache;

    while (ports[id].up()){
        while (local_cache.size() < local_size_max) {
```
6.3 Threading Models

Given the modular nature of the FFVM, the threading architecture of a virtual switch is incredibly flexible. In a single threaded architecture (STA), all component behavior, such as port work routines and application pipeline execution, is defined and executed by the main thread in the driver. To elevate an application to utilize multiple threads, the behavior for the desired components can be defined as a free function in the driver and assign that work stub to a thread.

6.3.1 Single Threaded

As a baseline implementation for most applications, an STA driver will service the FFVM and the server. The server port and FFVM ports can be tied to a polling
mechanism to switch between events occurring on multiple port objects. This architecture places the port and application pipeline processing on an even playing field. A process diagram for the STA wire driver is shown in Figure 6.1.

![Diagram of the Freeflow STA wire state machine](image)

Figure 6.1: The Freeflow STA wire state machine.
6.3.2 Thread Per Port

In a thread per port (TPP) model, the server and FFVM ports are separated to allow packet I/O to operate asynchronously. The main thread from the driver handles the acceptance of new connections to the server and spawns a new thread to execute the port work routine over them. Figure 6.2 illustrates the TPP wire driver process.

6.4 Results

The evaluation of the experiments conducted focuses on the measured performance of each application with respect to throughput and bandwidth. Throughput is defined as the number of packets per second (Pps) sent or received and bandwidth as the amount of data in Gigabits per second (Gbps) sent or received.

6.4.1 Endpoint

In the endpoint test, the average send and receive rates for throughput and bandwidth between a baseline (Flowcap to Flowcap) and the Freeflow (Flowcap to Endpoint) implementations are measured. This test helps measure the maximum rate that the FFVM can provide to a hosted application. Table 6.1 lists the observed performance for both of the testing scenarios and compares the Freeflow (FF) implementation with the baseline (Base).

The simple end-to-end connection created by the two Flowcap applications in the baseline example shows that the sink is able to process data faster than the source can send. Thus the rate at which packets can be received is bounded by the rate at
which they can be sent. Utilizing the Freeflow Endpoint application introduces some overhead, as a more complex processing model is used.
<table>
<thead>
<tr>
<th>Implementation</th>
<th>Pps Received</th>
<th>Pps Transmitted</th>
<th>Gbps Received</th>
<th>Gbps Transmitted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>1717502</td>
<td>1716152</td>
<td>8.468</td>
<td>8.461</td>
</tr>
<tr>
<td>Freeflow</td>
<td>627061</td>
<td>680367</td>
<td>3.091</td>
<td>3.323</td>
</tr>
<tr>
<td>FF/Base Δ</td>
<td>36.51%</td>
<td>39.64%</td>
<td>36.51%</td>
<td>39.27%</td>
</tr>
</tbody>
</table>

Table 6.1: Freeflow Endpoint and Flowcap throughput and bandwidth performance.

6.4.2 Wire

The Freeflow wire test evaluates the performance within the FFVM while hosting an application. For the purpose of this experiment the baseline case, the STA wire driver, is compared against the performance of the multiple versions of the TPP wire driver. Each TPP driver version corresponds to different shared resource strategies implemented. TPP-1 uses locked queues and copies the context on each push/pop, while TPP-2 utilizes lock-free queues. In TPP-3, lock-free queues hold containers of reusable buffer ids that are cached by each thread locally. Table 6.2 lists the observed performance for all four of the Freeflow wire implementations tested.

With each version of the TPP wire driver, the resulting performance of the FFVM hosted application changes. The initial TPP version resulted in a major slowdown with respect to packet throughput, but that issue was resolved by the use of lock-free queue structures in the second version. The final version implements more optimizations and results in the transmission rate exceeding the reception rate in terms of throughput and bandwidth.
Table 6.2: Freeflow wire driver performance metrics, including relative speedup.

6.5 Discussion

The results found in the previous section show that the FFVM is a bit behind in terms of performance. This can be attributed to numerous factors with the experimental setups in the execution environment. In each experiment run, all of the processes used are executed on the same device. System overhead from the operating system juggling multiple processes does not allow for optimal execution. This is obvious when comparing the results of the endpoint application, as the Flowcap processes are much more light weight and less burdensome on the OS.

Adding to this is the reliance on the Linux networking stack when using TCP (stream) sockets provided in the system API. Each call to the polling mechanism, send, and receive causes a context switch. Since this experiment emulates Ethernet
frames, the simple protocol that indicates the length of the frame warrants a peek and a read. Each packet that is received by the FFVM accrues three context switches and two for each send. This totals to five system calls, and as this experiment tests the system under heavy incoming traffic loads most of the time is spent in this calls. By context switching so frequently, the CPU suffers from thrashing and this causes a major slow down with respect to performance.

In order to boost performance in the FFVM, the implementation needs to be further towards the hardware end of the virtualization-materialization spectrum. Translating as much high level code down to native machine code provides the best execution environment for networking applications. For instance, flow tables can be crafted from hardware components, such as TCAMs, and manipulated by application logic. The fewer times the application-run time boundary is crossed, the more efficient the code will be executed.

However as we drift further towards a more realized implementation, we sacrifice portability for performance. This increases the amount of work that must be done by compilers to efficiently generate optimized native machine code. Additional concurrency models could also be supported by a shift towards the lower end of the spectrum, where new and exotic switch architectures sport cutting edge processors and accelerators.
SDN is still a relatively new networking architecture that is continuing to evolve. Much of the research in this domain is experimental, searching to establish the “correct” way to model basic network switch components, host compiled network applications, and efficiently abstract networking resources. The work presented in this thesis aims to find a balance between high level language support and optimal hardware execution through translation and abstraction, respectively. Contributions heavily revolve around abstracting low level networking hardware and processing network application instructions.

The FFVM provides a concrete implementation of an abstract network switch that provides the necessary resources required to efficiently host networking applications. An emphasis on modularity and re-configurability allows different virtual switches with varying capabilities, features, and resources to be provisioned and evaluated. Low level port abstractions give users control over networking interfaces while maintaining a high level of performance. High level network programming languages that can be compiled down to native binary files are able to be dynamically loaded and executed. The design and implementation of the FFVM is a culmination of the observed requirements found necessary to model an abstract network switch that is
fully programmable, is capable of hosting network applications, and provides access to networking hardware resources.

7.1 Future Work

The product of the contributions presented in this thesis is an initial implementation of a fully programmable virtual network switch. Additional work in this domain revolves around the two main ends of the virtualization-materialization spectrum: improved hosting support for more high level network programming languages and optimizations to hardware resource usage.

With respect to high level languages, support for the POF programming language would be easier as their models for decoding packet protocol header fields and forwarding behavior align with the FFVM context binding environment and flow table constructs. Other well established networking programming languages, such as P4, could also be considered to improve portability and application hosting capabilities.

Low level hardware abstraction aligns with the problems found in the domain of heterogeneous computing. Advances in that field can provide insight into future design and implementation strategies that could be incorporated into the FFVM. The usage of an IR allows for machine specific intrinsics to optimize the code generated by compilers to take full advantage of all computing resources available on a target system.
BIBLIOGRAPHY


