PVPP: A Programmable Vector Packet Processor

Sean Choi, Xiang Long[⋆], Muhammad Shahbaz[‡],
Skip Booth[◇], Andy Keep[◇], John Marshall[◇], Changhoon Kim[†]
Stanford University [⋆]Cornell University [‡]Princeton University [◇]Cisco, Inc [†]Barefoot Networks, Inc

Abstract

Recent work on simplifying data plane programming focuses on providing simple, high-level domain-specific languages (DSLs). These languages hide the complex and intricate details of the underlying switching substrate. Programmers write their data-plane programs in these languages which are then compiled to run on a given switch target, which further runs on a particular CPU architecture. However, the simplicity and the domain-specific nature of these DSLs and the lack of flexible switch interfaces that can be targeted by a DSL compiler restrict the ability to optimize generated code.

In this work, we present our findings on how the complexity of interfaces on a software switch target available to a compiler can affect the performance of compiled data plane programs. For our experiment platform, we built a P4 compiler called the *Programmable Vector Packet Processor (PVPP)* targeting the existing Vector Packet Processor (VPP) software switch. P4 is a data plane DSL based on match-action tables, while VPP uses a packet processing node graph model. PVPP compiles a data plane program written in P4 to VPP's internal graph representation. VPP also exposes a sophisticated interface for PVPP to interact with the various features of the underlying architecture *e.g.*, execution modes, memory types, and the batch I/O. Our evaluation shows that PVPP can efficiently exploit these features, resulting in the increased performance of the same data plane program.

CCS Concepts
•**Networks** → **Programmable networks;**

Keywords
Programmable Data Planes; Software Switch; VPP; P4; PVPP; Domain Specific Languages (DSL); Compiler Optimization

SOSR'17, April 3–4, 2017, Santa Clara, CA.

ISBN 978-1-4503-4947-5/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3050220.3060609

1 Introduction

There is ongoing interest in incorporating programmability into the data plane. One particular area of research is on building programmable software switches. A typical use case for software switches is in hypervisors where the network traffic routes through many virtual machines. The main advantage of using software switch is its ability to run on commodity CPUs, thus simplifying the development of network programs without relying on proprietary network processors.

An emerging approach in programming software switches is by expressing the data plane program through a domain-specific language (DSL), such as P4 [2], and then compiling it to a program that interacts with the interfaces available on the target software switch. However, there are challenges in compiling the data plane program efficiently, especially if the target switch exposes only a limited set of interfaces for specifying the runtime behavior, resulting in a limited ability for the compiler to add target-specific optimizations.

In this work, we argue that to improve the performance and flexibility of data plane programs compiled to software switches, the target switch must expose more complex interfaces that can interact with the underlying architecture beyond merely the interfaces for specifying high-level forwarding behavior. This would allow the compiler to fine-tune program performance in the target permeating down to the hardware level. To demonstrate this in action, we built the Programmable Vector Packet Processor (PVPP) as an extension of the Vector Packet Processing (VPP) framework [3]. VPP is a software switch target that exposes low-level primitives that can interact directly with the CPU. It also has a unique node graph packet processing model where a vector of packets is processed through a series of packet processing nodes each with separate instructions. The number of packet processing nodes or the amount of processing per node can be customized depending on the program being implemented. However, programming VPP is challenging for a programmer precisely due to its complex organization and the amount of manual fine-tuning that would be required to obtain acceptable performance. PVPP alleviates this by compiling a data plane program specified in P4 to code that executes within VPP. Thus, the programmer is allowed to express the desired data plane features through a friendly match-action tables abstraction, while the knowledge of how to effectively leverage VPP as a software switch target is shifted to PVPP.

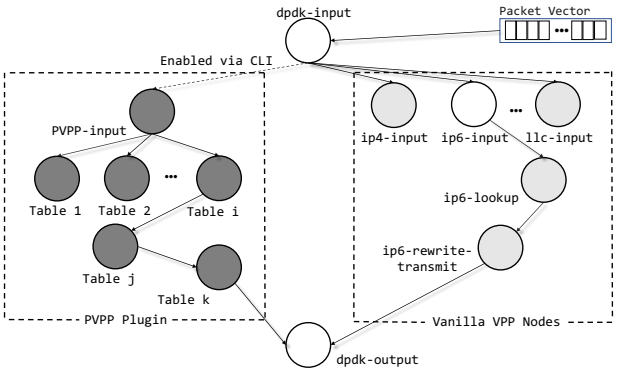

Figure 1: *VPP node graph structure with a plugin.*

Given PVPP, we were able to evaluate and analyze various optimizations by increasingly using more of the underlying architecture primitives exposed by VPP. These optimizations significantly improved throughput performance. This demonstrates that a more sophisticated interface exposed by the target software switch can actually enable more efficient compilation of data plane programs.

2 PVPP Architecture

VPP Plugins. The design of a P4-to-VPP compiler is governed by how the compiled code will ultimately integrate into the VPP infrastructure. VPP has a built-in plugin facility that allows a plugin to manipulate the node graph freely. A VPP plugin compiles as a dynamic library, and it declares its own graph nodes through an API that VPP exposes. PVPP compiles a data plane program to a VPP plugin, which allows the program to be loaded dynamically at runtime. An example of the relationship of a PVPP-compiled plugin with the entire packet processing node graph is shown in Figure 1.

P4-to-VPP Compiler. The compilation of P4-to-VPP code is a two-step process, each requiring an independent processing stage. The first stage uses the open-source P4 compiler [1], which takes the P4 program as input and generates an intermediate representation using the JavaScript Object Notation (JSON) format. In the second stage, we feed this intermediate representation to a custom JSON-to-VPP compiler to generate customized C code that further compiles to a VPP plugin.

3 Evaluation

We present our results to demonstrate the effect of various PVPP optimizations on its compiled program's throughput. The optimizations were only made possible by leveraging the underlying architecture exposed by VPP. The results show the potential benefits of added flexibility in the interface between the compiler and the target software switch.

We measured the performance of one of our benchmark applications, which consists of sending 64-byte packets to the switch whose PVPP-compiled program must perform an ethernet header match, TTL decrement, and finally writing the next hop destination MAC address. Table 1 shows the incremental improvements in throughput as each new optimization is applied. These measurements are only for the case where PVPP's entire packet processing is running on a single packet processing node. We intend to show the results for multiple-node processing, as well as the effect of dedicating multiple CPUs for PVPP in a follow-up work.

Optimization	Mpps	Increment (%)
Unoptimized	7.860	N/A
Reducing Tables	9.248	+1.388 (+17.70%)
Reducing Metadata	9.508	+0.260 (+02.81%)
Multiple Packet Processing	9.508	+0.000 (+00.00%)
Reducing Dereference	10.008	+0.425 (+04.43%)
Caching Interface Mapping	10.209	+0.201 (+02.01%)

Table 1: *Incremental improvements of each optimizations for PVPP*

The optimizations that we discuss in our evaluation include reducing the number of match-action tables processed by a node, reducing the amount of redundant metadata and metadata accesses, processing multiple packets in a single loop, reducing lower-level pointer dereferences, and caching the mapping of the logical port to the physical port. We see the greatest improvements by reducing the number of processing nodes and reducing the number of pointer dereferences as memory dereference operations require a large number of CPU cycles.

One interesting result is the improvements for multiple packet processing optimization. The purpose of processing a vector of nodes by VPP is to perform common tasks for many packets together to improve instruction cache locality. The multiple packet processing further improves this cache locality by unrolling the loop that iterates through the packet vector. Interestingly, as shown in Table 1, this optimization did not show any improvement for our simple application. This is mainly due to the small size of the instruction sets generated from our simple application, which fit entirely in the instruction cache in the CPU. Thus, given that VPP scales across multiple cores, we are planning to evaluate complex applications that reaps the benefits of processing multiple packets in a loop across many cores as part of our future work.

After the optimizations, the PVPP-compiled program's throughput is 10.209 Mpps. The throughput of an equivalent VPP's hand-tuned application is 10.748 Mpps, which is 5.25% higher. However, we must take into consideration that these libraries implement fixed protocols and have been optimized by hand. Nevertheless, there are various optimizations we are planning to implement in order to reduce this gap. These methods will appear as part of our future work.

References

[1] P4 compiler for the behavioral-model. `https://github.com/p4lang/p4c-bm`, 2015.

[2] 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. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3):87–95, July 2014.

[3] Vector Packet Processing Platform. `https://fd.io/technology`.