Selective and transparent acceleration of OpenFlow switches

Rolf Neugebauer — Netronome

Abstract

We present the design and implementation of a transparent and selective offload architecture, accelerating a mature software OpenFlow implementation (Open vSwitch) using a programmable network processor. Our design combines the flexibility offered by software OpenFlow implementations with the performance of specialised switch hardware. It is targeted at accelerating the emerging class of edge/gateway switches which are becoming increasingly important in some SDN designs. Our prototype implementation offers at least a 8-10x performance improvement over software only Open vSwitch for benchmarks where the majority of the packets can be handled by the network processor.

1 Introduction

The OpenFlow protocol was initially conceived to provide a simple abstraction over the datapath functionality provided by merchant switch chips to offer better programmability and enable innovation in the network infrastructure [6, 1]. OpenFlow is widely deployed, most significantly in the context of network management in data centres [11, 5]. OpenFlow networks generally consists of two components: Switches and controllers. In this paper we focus on OpenFlow switches.

OpenFlow switches come in two flavours: Software and hardware based implementations. Software switches, such as Open vSwitch [8], are typically feature complete but even mature implementations are often quite slow (see Section 3). While some techniques exist to improve the performance [4, 2], we agree with [3] that scaling to higher bandwidth will expose more fundamental bottlenecks. Hardware based OpenFlow switches provide line rate forwarding for large number of ports but lack the flexibility and feature completeness of software implementations. Even OpenFlow 1.0 features are rarely fully supported, primarily due to the limitations of the switches TCAMs and packet modification actions [7, 10]. This semantic gap is widening with the introduction of new OpenFlow specifications as well the proliferation of new tunneling protocols such as VxLAN, STT, and NV-GRE, used extensively in some SDN designs. Hybrid SDNs [2] have been proposed to bridge this gap, with a simple forwarding core and more intelligent edge switches; the latter to be implemented in software to enable flexibility and evolution at the edge.

In this paper we present an approach which offers the flexibility of a software only implementation with the performance of hardware implementations by selectively offloading (or accelerating) the datapath of a mature software switch implementation to a highly programmable network card. Packets which can be handled by the offload implementation will be processed entirely on the network card, while other packets are passed to the host CPU where they are processed as if they arrived via a standard NIC. Our implementation is based on Open vSwitch (OVS) [8] and uses Netronome’s NFP-32xx based NFE cards for the offload implementation. It outperforms the standard OVS implementation by a factor of 8-10x in some benchmarks while its operation is completely transparent to the end-user.

2 Design and implementation

An OpenFlow switch typically consists of the following components: a vPort module, mapping ingress and egress ports (or NICs) to some port abstraction, maintaining per port counters, and possibly implementing tunneling endpoints; a flow table (FT) which performs lookups on flow keys extracted from packet headers; and an action component, which performs a set of actions depending on the result of the flow table lookup. To enable selective offload, the implementation on the network card mirrors these components. The physical network ports of the network card still appear as normal NICs to the host and are used to deliver packets not handled by the offload. In addition to the components listed above, the offload implementation may also deploy an ingress filter to “short-circuit” particular types of packets, for example, based on protocol type.

The general flow of packets is depicted in Figure 1 with packets flowing from left to right and the vPort and NIC components split into ingress and egress. Incoming packets are subjected to a simple filter and, if filtered out, are passed to the host. Packets are then associated with an ingress vPort based on physical ingress port and tunnel headers, which are stripped, if present. If a tunnel header cannot be matched, the packet is passed to the host. Next, the flow key extracted from the packet header is looked up in a flow table (FT) on the device. If a matching entry is found, the associated actions are performed, the packet is subjected to any egress vPort actions (stats, tunnel), and is transmitted. Packets with no FT entry are sent to the host. Any packets sent to the host are handled by the switch implementation there.

![Figure 1: Packet flow with selective offload](image-url)
The OpenFlow switch software on the host needs to be augmented to manage and control the behaviour of the offload implementation. This typically involves deciding which flow table entries can and should be offloaded to the device (based on the implemented feature set) and to sync statistics from the device to the host.

Our implementation is based on Open vSwitch 1.7. With OVS, the core functionality is implemented in userspace with an in-kernel fast-path acting as a flow-based cache for forwarding decisions made by the userspace code. Our offload implementation mirrors the OVS kernel fastpath and selectively offloads some of its processing. The offload is managed by 12 hooks in the standard OVS kernel module and is implemented in a separate kernel module. While this kernel module is specific to our offload solution, the hooks themselves are kept generic. The offload is completely transparent to the OVS user-space daemons and tools: no modifications were necessary and the offload is simply enabled by loading the offload kernel module.

We implemented our prototype on a PCIe card based on Netronome’s NFP-32xx processor, a 40 core, multi-threaded network flow processor. Our implementation currently supports exact matches on the full OVS IPv4 flow key (a 14-tuple), implements all packet header modification actions, offers GRE based tunnel vPorts, and supports the full set of vPort and flow statistics. The flow table supports 1M entries. It requires 150MB of memory and can be extended as up to 4GB are available.

The current implementation supports a significant and commonly used subset of the OVS feature set which can easily be further extended as needed. The offload application on the NFP-32xx is written in a variant of C and only consists of around 1000 lines of code (LOC, measured with cloc), with the core of the functionality being implemented with 600 LOC. The kernel module, implementing the hook functions, consists of 1500 LOC.

3 Preliminary performance results

We evaluated our prototype with a number of benchmarks in a forwarding configuration. Our implementation easily outperforms a configuration with standard Intel 82599EB 10GigE NICs on the same system (a 2.3GHz, six core Intel Xeon E5-2630). We can forward traffic of all frame sizes from 128B and larger at 10GigE line rate, corresponding to 8.44Mpps (Million packets per second), irrespective of the number and type of header modification actions configured. Our implementation also supports optional GRE encapsulation at these rates and scales to large number of flows (up to 400k concurrent flows tested). We expect to be able to process packets at up to $2 \times 10\text{GigE}$ line rate, but our benchmarks currently can not generate traffic at these rates. In contrast, using the Intel NICs, the same system can only forward around 1.1Mpps irrespective of packet size. This is further reduced to around 650Kpps and 830Kpps when the host also performs GRE decapsulation and encapsulation respectively. The host CPU is the bottleneck in these tests while it is mostly idle with our implementation.

More realistic workloads require flow table updates as new connections arrive. An extreme case is represented by ovs-benchmark, which creates TCP connections and tears them down immediately. Standard OVS achieves around 32.9K new connections/s on the test system while our implementation sustains around 35.1K new connections/s. In this benchmark, all packets are processed by the host CPU in both configurations and it is the bottleneck. The slight difference between the two configurations is likely due to the differences in drivers and the NIC implementation.

4 Conclusion

We presented the design and implementation of a selective offload architecture for OpenFlow switches and demonstrated that this approach can yield significant performance improvements with a moderate programming effort. While some hardware based switches use a similar approach, we believe that highly programmable network cards in combination with OpenFlow switch implementations on general purpose CPUs provide a better, more flexible alternative for the emerging class of edge/gateway switches in some SDN designs.

References