Converging Approaches in Software Switches

Ben Pfaff

June 19, 2016

1 Converging Approaches in Software Switches

I feel a little strange standing in front of a group with so many academics and presenting something that is not a research talk. Just to make that clear, this talk is not a research talk and I don't know whether the material will ever evolve into a research paper. I don't even have any good numbers here. There *are* going to be two graphs, but the numbers in the graphs are entirely made up.

2 The Point

I don't want to keep you hanging, so I'm going to start by getting right to the point.

There are two main ways to build software switch pipelines: "code-driven" and "data-driven." Usually, these are considered to be alternatives. I am going to argue that they are actually complementary.

These terms "code-driven" and "data-driven" are ones I just made up myself. I'll define them in a minute, so don't tune out yet.

3 Irrelevancies

I'm going to spend the rest of the talk going into details on my main point, but before that I also want to make it clear what I'm not talking about. There's been a lot of talk in the last few years about packet I/O methods, by which I mean ways of getting packets in and out of physical interfaces and, increasingly, in and out of VMs and containers. Fast packet I/O is essential to performance but it's barely relevant to switch pipeline designs. In fact, Open vSwitch supports all the packet I/O methods listed here, although the netmap support was on an old branch that hasn't ever been integrated.

4 Code-Driven Switch Pipeline

Here's the first kind of software switch pipeline that I want to talk about, a "code-driven" pipeline. In a code-driven pipeline, for each packet, the switch executes a series of code fragments, which I'm going to call "stages." Most code-driven switches allow stages to be chained together linearly or with branching and maybe even iteration.

Maybe some of you are saying "duh" at this point, because this is a really obvious way to build a software switch, and lots of switches have been built this way.

The stages in a code-driven switch pipeline tend to be very loosely coupled. They're often implemented as something close to a function that takes a packet as input and produces a packet as output. This makes it really easy to extend this kind of a switch, and even to incorporate third party code as plugins.

5 Code-Driven Pipeline Stages

Let's zoom in on one of the stages in a code-driven switch and take a look at its properties.

Because it's implemented as a piece of code, it can do pretty much anything, or it can do nothing at all, or anything in between.

The code inside the stage is a black box, which means that there's no way at a high level to optimize a sequence of stages. Thus, as you can see in my graph of made-up numbers, the per-packet latency steadily increases as the number of stages increases.

On the other hand, there's not much work for the code-driven switch to do beyond packet I/O and calling into the stages, so there's very little fixed overhead, which you can see from the latency in the graph when there are 0 stages.

This means that an empty pipeline, one that just copies packets from an ingress port to an egress port, runs very fast in a code-driven switch. That makes such a benchmark very attractive to authors of such switches.

6 DPDK and Netmap Are Not Software Switches

I'm going to talk about some examples of code-driven software switches in a minute, but before I get to that I want to re-emphasize the difference between packet I/O libraries and software switches.

DPDK and Netmap are packet I/O libraries, not software switches. In both cases, however, early publications about them emphasized their performance as "switches," that is, when they were used as an expensive crossover cable between Ethernet ports, and compared performance against actual software switches.

For example, my recollection, which might be inaccurate, is that some early reports on DPDK performance compared the performance of unmodified Open vSwitch that used the OVS kernel module, against a version of Open vSwitch

that had been modified to use DPDK *and in which all of the switching logic had been deleted and replaced by a hard-coded loopback-to-ingress-port stage.*

A fairer comparison would have been against other packet I/O methods.

7 Some Code-Driven Switches

Let me tell you about a few code-driven switches. These are roughly in the chronological order of their creation.

Linux bridge This is the oldest example. It was originally designed as a really simple Ethernet MAC-learning switch but there has been feature creep over time. It tends to be slow because Linux is optimized for delivery to sockets rather than switching.

Click Click is the classic academic reference for this kind of switch pipeline. It has a reputation for being slow by today's standards. My guess is that performance was not a priority in its design, or that what's important to achieve performance has shifted significantly since it was built.

VMware VDS This is VMware's oldest switch that is still in mainstream use. It follows the code-driven pipeline model to a tee by imposing stages that it called "I/O chains" between physical and virtual ports. It would be a classic example except that it's closed source. It is, however, open enough to plugins from VMware partners that Cisco was able to replace it with Cisco's 1000V software switch.

VMware NSX Edge Probably most of you have never heard of this switch. It's based on DPDK. It's been under development for about two years as part of the NSX-T product that VMware just released. The code is completely separately developed from Open vSwitch and from VDS, which means that VMware does really develop three completely independent software switch code bases. In my opinion, this is a mistake, but no one asked me.

VPP This is a DPDK-based switch that Cisco recently released as open source. The main interesting aspect of its code is an innovative batching technique, at least according to a Cisco engineer I cornered at a drunken party at the last OpenStack summit.

BESS Even though BESS stands for "Berkeley Extensible Software Switch," its webpage claims that it is not a virtual switch. This is a confusing distinction that I do not understand, but at any rate it *can* act as a switch and it is DPDK based. It seems to be comparable to a modern version of Click optimized for performance.

8 Data-Driven Switch Pipeline

I've talked a lot about code-driven switches. Now it's time to describe data-driven switch pipelines.

A data-driven pipeline has a single block of code, instead of many, as an engine that drives each packet through a series of stages. The stages contain data tables instead of code blocks, but they can be chained together in the same way as a code-driven switch. A data-driven pipeline also adds a parser step at the beginning of the pipeline, which pulls headers out of the packet into a form convenient for the code engine.

At first glance, a data-driven pipeline doesn't have much to recommend it. Programmers like writing code and at first feel straitjacketed by the need to implement their programs in terms of data tables. The code engine limits the switch's capabilities in other ways too, since the switch can only handle the protocols and actions that are built into the engine. Doing parsing once at the beginning of the pipeline does tend to be faster than doing it once per stage as is typical in a code-driven design, but that's a mixed bag too because usually the parser will pull out every protocol it supports even if none of the stages actually cares about that protocol, and that can cost some performance.

9 Data-Driven Pipeline Stages

There is, however, an important advantage of a data-driven pipeline. That is that the engine has insight into the whole pipeline, like a compiler or an interpreter does into a program. It can act as an optimizer. Most importantly, it can take the crossproduct of many stages, in a lazy way, and turn many table lookups into just a single table lookup. This means that increasing the number of stages doesn't increase the per-packet forwarding latency very much, often not at all, as long as the caching techniques work out.

So, a pipeline with many stages isn't much slower than one with few stages. But there's a high fixed cost for the parser itself. You can see what I mean on my made-up graph here: the code-driven switch is fast with few stages, and the data-driven switch is fast with many stages.

Benchmarks tend to use simple pipelines with few stages, especially when they're being run by people showing off the performance of a code-driven switch.

There's another valuable feature of data-driven pipelines. It's pretty easy to take advantage of the features of the NICs that are starting to get into servers to offload packet classification. It's not complete offload, but it's often enough to eliminate matching possibilities and significantly speed up data-driven switches. This isn't mainstream yet but I'm still hoping that it will become common; real commodity hardware does support it. I don't know how this feature could be used in code-driven switches to the same effect.

10 Some Data-driven Switches

Here are the data-driven switch I know about. There are obviously not very many of them.

Open vSwitch This is what I know best and will talk about most. Its design was driven by the needs of network virtualization, that is, software like VMware NSX or OVN. Network virtualization is a sophisticated application that needs several stages, typically at least 15.

MidoNet From Midokura. MidoNet is also designed for network virtualization, but it is an application more than a platform. It uses Open vSwitch kernel module for packet I/O, but its userspace is completely different.

11 Crossover

The question now is whether we can combine the strengths of code-driven and data-driven software switch pipelines.

To review:

The features of code-driven switches we'd want are the low fixed per-packet overhead and the flexibility of a loosely coupled plugin model.

The features of data-driven switches we'd want are the Low per-stage overhead and the common parser.

I don't have a complete answer. I'll talk about some trends I see.

12 Code-Driven Moving Toward Data-Driven

Perhaps you are skeptical about the strengths I've claimed for data-driven pipelines. You might be saying, "I think that if a data-driven pipeline is faster than a code-driven one, for the same application, then the code-driven pipeline code is badly written."

Maybe you are right. I do not know how to evaluate whether code is written in the fastest possible way.

But I have two cases to bring up from inside VMware. As I've said, VMware separately develops at least three software switches, and two of them are code-driven. Both of them are tending to move in a data-driven direction.

VMware VDS. The VDS code-driven pipeline was fast enough for relatively simple applications. When VMware added network virtualization functionality for NSX-T, all the new stages added a lot of overhead, even though they were well written. Good engineers spent a lot of time optimizing them, but in the end they started working on a data-driven approach that was more cacheable. I don't know whether this approach has been adopted yet.

As an aside, there's other work going on in VDS to switch from its home-grown packet I/O to DPDK. That's independent and orthogonal to whether the pipeline is code-driven or data-driven. It's also unusual in that DPDK is being used in the ESX kernel instead of its userspace.

VMware NSX Edge switch. The NSX Edge switch project was started specifically to build a new software switch that would be application-specific to network virtualization. The team built it on top of DPDK in a code-driven fashion. They knew exactly what application they needed to implement and had no legacy code or protocols to support, so you'd expect that it would be blazing fast.

I have heard that it is not. All the stages slow it down.

I understand that the team has been working on data-driven style optimizations.

13 Data-Driven Moving Toward Code-Driven

I mentioned two strengths of code-driven switches that would be nice to have in a data-driven switch.

13.1 Fixed Per-Packet Overhead

The first is low fixed per-packet overhead. There are two sources of high fixed overhead in data-driven models that we can choose to attack.

The first is the cost of parsing. This exists because current data-driven switches don't know what fields and protocols will be used at runtime, so they have fixed parsers that always extract every field. The fix is to allow the controller to specify the protocols that will actually be used, using a domain-specific language such as P4 or OpenFlow Table Type Patterns.

The second is the cost of classification. This can be reduced via hardware offload, as I've already talked about. I think that this will really happen in the next year or two.

13.2 Flexibility

The other strength that would be nice to have is flexibility. It's harder to extend data-driven pipelines, specifically to implement new matches and actions.

However, OVS is moving to increase flexibility in a number of ways.

First, arbitrary code could be integrated into actions if there were a reasonable way to specify it. Fortunately, there are two technologies coming down the pike that allow for this in different ways. There's eBPF, which is is a low-level bytecode, conceptually similar to JVM bytecode. These days the Linux kernel allows userspace programs to install eBPF into the kernel. This could be used to specify actions and other code. There's also P4, which in addition to specifying parsers, could specify actions. It's likely that P4 would just be a convenient

domain-specific language for networks and that in fact under the covers this would get compiled down to eBPF anyway.

OVS is also starting to add specialized support for invoking features of the Linux kernel networking stack that users want. These include TCP connection tracking, which is useful for firewalls, and NAT.

Finally, SoftFlow, by Ethan Jackson at Berkeley, shows how to integrate OVS comfortably and naturally into a pipeline of various kinds of middleboxes. SoftFlow will be presented this Wednesday at USENIX ATC in Denver.

14 Conclusion

Two seemingly different approaches to software switch pipelines, which I call "code-driven" and "data-driven," may prove to converge closer that one would initially expect.