Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization

Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat, *Google, Inc.*

https://www.usenix.org/conference/nsdi18/presentation/dalton

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18).

April 9–11, 2018 • Renton, WA, USA

ISBN 978-1-939133-01-4

Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization

Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs,
Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert,
Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis,
Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter,
Uday Naik, and Amin Vahdat
Google, Inc.

Abstract

This paper presents our design and experience with Andromeda, Google Cloud Platform's network virtualization stack. Our production deployment poses several challenging requirements, including performance isolation among customer virtual networks, scalability, rapid provisioning of large numbers of virtual hosts, bandwidth and latency largely indistinguishable from the underlying hardware, and high feature velocity combined with high availability.

Andromeda is designed around a flexible hierarchy of flow processing paths. Flows are mapped to a programming path dynamically based on feature and performance requirements. We introduce the *Hoverboard* programming model, which uses gateways for the long tail of low bandwidth flows, and enables the control plane to program network connectivity for tens of thousands of VMs in seconds. The on-host dataplane is based around a high-performance OS bypass software packet processing path. CPU-intensive per packet operations with higher latency targets are executed on coprocessor threads. This architecture allows Andromeda to decouple feature growth from fast path performance, as many features can be implemented solely on the coprocessor path. We demonstrate that the Andromeda datapath achieves performance that is competitive with hardware while maintaining the flexibility and velocity of a software-based architecture.

1 Introduction

The rise of Cloud Computing presents new opportunities and challenges for networking. Cloud providers must support virtual networks with high performance and a rich set of features such as load balancing, firewall, VPN, QoS, DoS protection, isolation, and NAT, all while operating at a global scale. There has been substantial research in network support for Cloud Computing, in particular in high-speed dataplanes [17, 34], virtualized routing infrastructure [6, 22, 23, 31], and NFV middleboxes [14, 16]. Typical research efforts focus on point problems in the space, rather than the challenges of bringing a working system together end to end. We developed *Andromeda*, the network virtualization environment for Google Cloud Platform (GCP). We use this experience to show how we divide functionality across a global, hierarchical control plane, a high-speed on-host virtual switch, packet processors, and extensible gateways.

This paper focuses on the following topics:

- The Andromeda Control plane is designed for agility, availability, isolation, and scalability. Scale up and down of compute and rapid provisioning of virtual infrastructure means that the control plane must achieve high performance and availability. Andromeda scales to networks over 100,000 VMs, and processes network updates with a median latency of 184ms. Operations on behalf of one virtual network, e.g., spinning up 10k VMs, should not impact responsiveness for other networks.
- The Andromeda Dataplane is composed of a flexible set of flow processing paths. The *Hoverboard path* enables control plane scaling by processing the long tail of mostly idle flows on dedicated gateways. Active flows are processed by our on-host dataplane. The on-host *Fast Path* is used for performance-critical flows and currently has a 300ns per-packet CPU budget. Expensive per-packet work on-host is performed on the *Coprocessor Path*. We found that most middlebox functionality such as stateful firewalls can be implemented in the on-host dataplane. This improves latency and avoids the high cost of provisioning middleboxes for active flows.
- To remain at the cutting edge, we constantly deploy new features, new hardware, and performance improvements. To maintain high deployment velocity without sacrificing availability, Andromeda supports transparent VM live migration and non-disruptive dataplane upgrades.

We describe the design of Andromeda and our experience evolving it over five years. Taken together, we have improved throughput by 19x, CPU efficiency by 16x, latency by 7x, and maximum network size by 50x, relative to our own initial production deployment. Andromeda also improved velocity through transparent VM migration and weekly non-disruptive dataplane upgrades, all while delivering a range of new end-customer Cloud features.

2 Overview

2.1 Requirements and Design Goals

A robust network virtualization environment must support a number of baseline and advanced features. Before giving an overview of our approach, we start with a list of features and requirements that informed our thinking:

- At the most basic level, network virtualization requires supporting isolated virtual networks for individual customers with the illusion that VMs in the virtual network are running on their own private IP network. VMs in one virtual network should be able to communicate with one another, to internal Cloud provider services, to third party providers, and to the Internet, all subject to customer policy while isolated from actions in other virtual networks. While an ideal, our target is to support the same throughput and latency available from the underlying hardware.
- Beyond basic connectivity, we must support constantly evolving network features. Examples include billing, DoS protection, tracing, performance monitoring, and firewalls. We added and evolved these features and navigated several major architectural shifts, such as transitioning to a kernel bypass dataplane, all without VM disruption.
- A promise of Cloud Computing is higher availability than what can be provisioned in smaller-scale deployments. Our network provides global connectivity, and it is a core dependency for many services; therefore, it must be carefully designed to localize failures and meet stringent availability targets.
- Operationally, we have found live virtual machine migration [5, 11, 18, 30] to be a requirement for both overall availability and for feature velocity of our infrastructure. Live migration has a number of stringent requirements, including packet delivery during the move from one physical server to another, as well as minimizing the duration of any performance degradation.
- Use of GCP is growing rapidly, both in the number of virtual networks and the number of VMs per network. One critical consideration is *control plane scalability*. Large networks pose three challenges, relative to small networks: they require larger routing tables, the routing tables must be disseminated more broadly, and they tend to have higher rates of churn. The control plane must be able to support networks with tens or even hundreds of thousands of VMs. Additionally, low programming latency is important for autoscaling and failover. Furthermore, the ability to provision large networks quickly makes it possible to run large-scale workloads such as MapReduce inexpensively, quickly, and on demand.

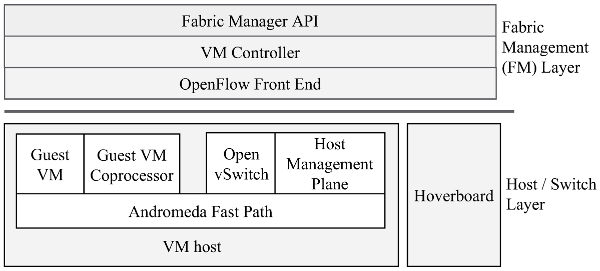


Figure 1: Andromeda Stack

2.2 Design Overview

Our general design approach centers around hierarchical data and control planes. The control plane is designed around a global hierarchy coupled with the overall Cloud cluster management layer. For example, configuring Andromeda is only one step among many in configuring compute, storage, access control, etc. For isolation, we run separate control stacks in every *cluster*. A cluster is a collection of colocated machines with uniform network connectivity that share the same hardware failure domain. The Andromeda control plane maintains information about where every VM in the network currently runs, and all higher-level product and infrastructure state such as firewalls, load balancers, and routing policy. The control plane installs selected subsets of this state in individual servers through a hierarchy of controllers.

The dataplane consists of a set of flexible user-space packet processing paths. The VM host *Fast Path* is the first path in the dataplane hierarchy and targets raw performance over flexibility. The Fast Path has a per-packet CPU budget of 300ns. Achieving this goal requires limiting both the complexity of Fast Path work and the amount of Fast Path state required to process packets. High performance, latency-critical flows are processed end to end on the Fast Path. Andromeda forwards other flows from the Fast Path to *Hoverboards* or *Coprocessors* for additional processing. On-host software Coprocessors running in per-VM floating threads perform per-packet work that is CPU-intensive or without strict latency targets. Coprocessors decouple feature growth from Fast Path performance, providing isolation, ease of programming, and scalability.

Andromeda sends packets that do not match a flow rule on the VM host to Hoverboards, dedicated gateways that perform virtual network routing. The control plane dynamically selects only active flows to be installed on VM hosts based on current communication patterns. Hoverboards process the long tail of mostly idle flows. Since typically only a small subset of possible VM pairs in a network communicate, only a small fraction of network configuration is required at an individual VM host. Avoiding the need to install full forwarding information on every host improves per-server memory utilization and control-plane CPU scalability by over an order of magnitude.

3 Control Plane

The Andromeda control plane consists of three layers:

Cluster Management (CM) Layer: The CM layer provisions networking, storage, and compute resources on behalf of users. This layer is not networking-specific, and is beyond the scope of this paper.

Fabric Management (FM) Layer: The FM layer exposes a high-level API for the CM Layer to configure virtual networks. The API expresses user intent and abstracts implementation details, such as the mechanism for programming switches, the encapsulation format, and the network elements responsible for specific functions.

Switch Layer: In this layer, two types of software switches support primitives such as encapsulation, forwarding, firewall, and load balancing. Each VM host has a virtual switch based on Open vSwitch [33], which handles traffic for all VMs on the host. Hoverboards are standalone switches, which act as default routers for some flows.

3.1 FM Layer

When the CM layer connects to a FM controller, it sends a `full` update containing the complete FM configuration for the cluster. Subsequent updates are diffs against previously sent configuration. The FM configuration consists of a set of *entities* with a known type, a unique name, and parameters defining entity properties. Figure 2 lists some examples of FM entities.

The FM API is implemented by multiple types of controllers, each responsible for different sets of network devices. Presently, VM Controllers (VMCs) program VM hosts and Hoverboards, while Load-Balancing Controllers [12] program load balancers. This paper focuses on VMCs.

VMCs program VM host switches using a combination of OpenFlow [3, 28] and proprietary extensions. VMCs send OpenFlow requests to proxies called *OpenFlow Front Ends (OFEs)* via RPC – an architecture inspired by Onix [25]. OFEs translate those requests to OpenFlow. OFEs decouple the controller architecture from the OpenFlow protocol. Since OFEs maintain little internal state, they also serve as a stable control point for VM host switches. Each switch has a stable OFE connection without regard for controller upgrade or repartitioning.

OFEs send switch events to VMCs, such as when a switch connects to it, or when virtual ports are added for new VMs. VMCs generate OpenFlow programming for switches by synthesizing the abstract FM programming and physical information reported in switch events. When a VMC is notified that a switch connected, it reconciles the switch's OpenFlow state by reading the switch's state via the OFE, comparing that to the state expected by the VMC, and issuing update operations to resolve any differences.

Network: QoS, firewall rules, ...
VM: Private IP, external IPs, tags, ...
Subnetwork: IP prefix
Route: IP prefix, priority, next hop, ...

Figure 2: Examples of FM Entities

Multiple VMC partitions are deployed in every cluster. Each partition is responsible for a fraction of the cluster hosts, determined by consistent hashing [20]. The OFEs broadcast some events, such as switch-connected events, to all VMC partitions. The VMC partition responsible for the host switch that generated the event will then subscribe to other events from that host.

3.2 Switch Layer

The switch layer has a programmable software switch on each VM host, as well as software switches called Hoverboards, which run on dedicated machines. Hoverboards and host switches run a user-space dataplane and share a common framework for constructing high-performance packet processors. These dataplanes bypass the host kernel network stack, and achieve high performance through a variety of techniques. Section 4 discusses the VM host dataplane architecture.

We employ a modified Open vSwitch [33] for the control portion of Andromeda's VM host switches. A user-space process called *vswitchd* receives OpenFlow programming from the OFE, and programs the datapath. The dataplane contains a flow cache, and sends packets that miss in the cache to vswitchd. vswitchd looks up the flow in its OpenFlow tables and inserts a cache entry.

We have modified the switch in a number of substantial ways. We added a C++ wrapper to the C-based vswitchd, to include a configuration mechanism, debugging hooks, and remote health checks. A management plane process called the *host agent* supports VM lifecycle events, such as creation, deletion, and migration. For example, when a VM is created, the host agent connects it to the switch by configuring a virtual port in Open vSwitch for each VM network interface. The host agent also updates VM to virtual port mapping in the FM.

Extension modules add functionality not readily expressed in OpenFlow. Such extensions include connection tracking firewall, billing, sticky load balancing, security token validation, and WAN bandwidth enforcement [26]. The extension framework consists of upcall handlers run during flow lookup. For example, *Pre-lookup handlers* manage flow cache misses prior to OpenFlow lookup. One such handler validates security tokens, which are cryptographic ids inserted into packet headers to prevent spoofing in the fabric. Another type is *group lookup handlers*, which override the behavior of specific OpenFlow groups, e.g., to provide sticky load balancing.

3.3 Scalable Network Programming

A key question we faced in the design and evolution of Andromeda was how to maintain correct forwarding behavior for individual virtual networks that could in theory scale to millions of individual VMs. Traditional networks heavily leverage address aggregation and physical locality to scale the programming of forwarding behavior. Andromeda, in contrast, decouples virtual and physical addresses [23]. This provides many benefits, including flexible addressing for virtual networks, and the ability to transparently migrate VMs within the physical infrastructure (Section 3.4). However, this flexibility comes at a cost, especially with respect to scaling the control plane.

One of the following three models is typically used to program software-defined networks:

Preprogrammed Model: The control plane programs a full mesh of forwarding rules from each VM to every other VM in the network. This model provides consistent and predictable performance. However, control plane overhead scales quadratically with network size, and any change in virtual network topology requires a propagation of state to every node in the network.

On Demand Model: The first packet of a flow is sent to the controller, which programs the required forwarding rule. This approach scales better than the preprogrammed model. However, the first packet of a flow has significantly higher latency. Furthermore, this model is very sensitive to control plane outages, and worse, it exposes the control plane to accidental or malicious packet floods from VMs. Rate limiting can mitigate such floods, but doing so while preserving fairness and isolation across tenants is complex.

Gateway Model: VMs send all packets of a specific type (e.g., all packets destined for the Internet) to a gateway device, designed for high speed packet processing. This model provides predictable performance and control plane scalability, since changes in virtual network state need to be communicated to a small number of gateways. The downside is that the number of gateways needs to scale with the usage of the network. Worse, the gateways need to be provisioned for peak bandwidth usage and we have found that peak to average bandwidth demands can vary by up to a factor of 100, making it a challenge to provision gateway capacity efficiently.

3.3.1 Hoverboard Model

Andromeda originally used the preprogrammed model for VM-VM communication, but we found that it was difficult to scale to large networks. Additionally, the preprogrammed model did not support *agility* – the ability to rapidly provision infrastructure – which is a key requirement for on-demand batch computing.

To address these challenges, we introduced the *Hoverboard Model*, which combines the benefits of On-Demand and Gateway models. The Andromeda VM host stack sends all packets for which it does not have a route to Hoverboard gateways, which have forwarding information for all virtual networks. However, unlike the gateway model, the control plane dynamically detects flows that exceed a specified usage threshold and programs *offload flows*, which are direct host-to-host flows that bypass the Hoverboards. Figure 3 shows flows that use Hoverboards as a default router and flows for which the control plane has programmed a direct host to host route and *offloaded* them from the Hoverboard.

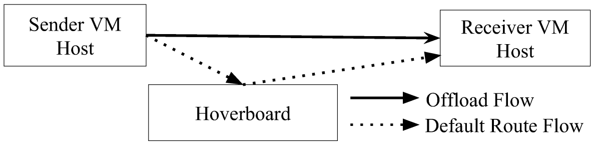


Figure 3: Hoverboard Packet Forwarding

The control plane detects these high bandwidth flows based on usage reports from the sending VM hosts. For robustness, we do not rely on usage reports from the Hoverboards themselves: the Hoverboards may not be able to send such reports if they are overloaded, and thus the control plane would be unable to install offload flows to reduce the load.

The Hoverboard model avoids the pitfalls with the other models. It is scalable and easy to provision: Our evaluation (see Section 5.3) shows that the distribution of flow bandwidth tends to be highly skewed, so a small number of offload flows installed by the control plane diverts the vast majority of the traffic in the cluster away from the Hoverboards. Additionally, unlike the On Demand Model, all packets are handled by a high performance datapath designed for low latency.

Currently we use the Hoverboard model only to make routing scale, but we plan to extend Hoverboards to support load balancing and other middlebox features. Stateful features, such as firewall connection tracking or sticky load balancing, are more challenging to support in the Hoverboard model. Challenges include state loss during Hoverboard upgrade or failure [21], transferring state when offloading, and ensuring that flows are "sticky" to the Hoverboard that has the correct state.

3.4 Transparent VM Live Migration

We opted for a high-performance software-based architecture instead of a hardware-only solution like SR-IOV because software enables flexible, high-velocity feature deployment (Section 4.1). VM Live Migration would be difficult to deploy transparently with SR-IOV as the guest would need to cope with different physical NIC resources on the migration target host.

Live migration [5, 11] makes it possible to move a running VM to a different host to facilitate maintenance, upgrades, and placement optimization. Migrations are

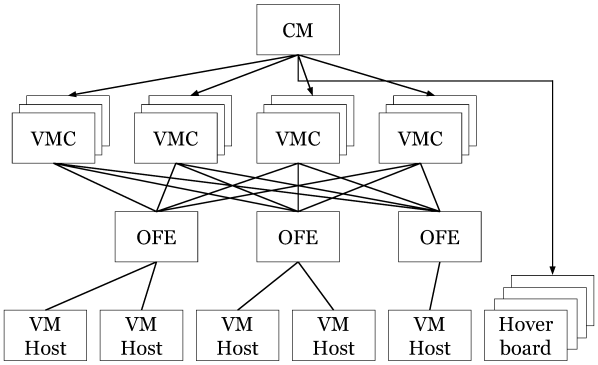


Figure 4: Control Plane Replication and Partitioning

virtually transparent to the VM: the VM continues to see the same virtual Ethernet device, and Andromeda ensures that network connections are not interrupted. The VM is paused during the migration blackout phase, which has a median duration of 65ms and 99th percentile of 388ms. After blackout, the VM resumes execution on the destination host.

Prior work [11, 30] focuses on migrations within a single layer-2 domain. In contrast, Andromeda supports global virtual networks and migrations across clusters. A key challenge is avoiding packet loss during migration even though global routing tables cannot be updated instantly. During blackout, Andromeda enables *hairpin* flows on the migration source host. The migration source will hairpin any ingress packets intended for the migrating VM by forwarding the packets to the migration destination. After blackout ends, other VM hosts and Hoverboards are updated to send packets destined for the migrated VM to the migration destination host directly. Finally, the hairpin flows on the migration source are removed. VROOM [35] uses a similar approach to live migrate virtual routers.

3.5 Reliability

The Andromeda control plane is designed to be highly available. To tolerate machine failures, each VMC partition consists of a Chubby-elected [9] master and two standbys. Figure 4 shows an Andromeda instance for a cluster with four replicated VMC partitions. We found the following principles important in designing a reliable, global network control plane:

Scoped Control Planes: Andromeda programs networks that can be global in scope, so the cluster control plane must receive updates for VMs in all other clusters. We must ensure that a bad update or overload in one region cannot spill over to the control planes for other regions. To address this challenge, we split the control plane into a *regionally aware control plane (RACP)* and a *globally aware control plane (GACP)*. The RACP programs all intra-region network connectivity, with its configuration limited to VMs in the local region. The GACP manages inter-region connectivity, receiving FM updates for both local and remote regions. This approach ensures that intra-region networking in each region is a separate failure domain.

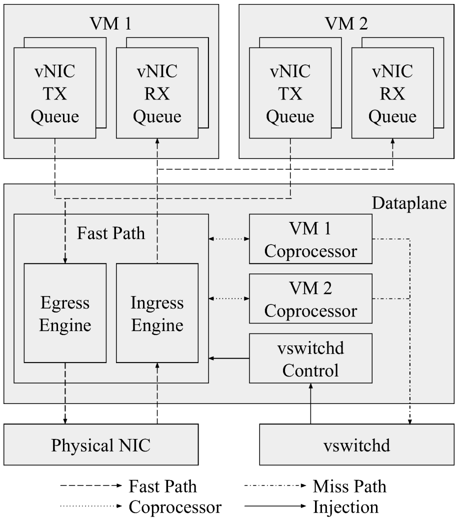


Figure 5: Host Dataplane Overview

Network Isolation: Churn within one customer's network should not impact network programming latency for other networks. To that end, VMCs maintain separate queues for processing FM updates for each network.

Fail static: Every layer of the control plane is designed to fail static. For example, hosts continue to forward packets using the last-known-good programming state, even if VMCs are unavailable. Hosts checkpoint their state to allow them to preserve the fail static behavior across host process restarts, and we routinely test fail-static behavior by simulating outages in a test environment.

4 VM Host Dataplane

Figure 5 illustrates the Andromeda VM host dataplane. The dataplane is a userspace process that performs all on-host VM packet processing, combining both virtual NIC and virtual switch functionality. There are two primary dataplane packet processing paths: the *Fast Path* and the *Coprocessor Path*. The Fast Path performs high-performance packet processing work such as encapsulation and routing via a flow table. The Coprocessor Path performs packet work that is either CPU-intensive or does not have strict latency requirements, such as WAN packet encryption.

Each VM is managed by a userspace Virtual Machine Manager (VMM). There is one VMM per VM. The VMM sends RPCs to the Andromeda dataplane for operations such as mapping guest VM memory, configuring virtual NIC interrupts and offloads, and attaching virtual NIC queues.

The Fast Path maintains a cache of forwarding state and associated packet processing actions. When a packet misses in the Fast Path cache, it is sent to on-host vswitchd, which maintains the full forwarding state programmed by VMC. Vswitchd sends flow cache update instructions and *reinjects* packets into the Fast Path.

A key goal of the Fast Path is to provide high-throughput, low-latency VM networking. For performance, the Fast Path busy polls on a dedicated logical CPU. The other logical CPU on the physical core runs low-CPU control plane work, such as RPC processing, leaving most of the physical core for Fast Path use. The Fast Path can process over three million small packets per second with a single CPU, corresponding to a per-packet CPU budget of 300ns. The Fast Path can be scaled to multiple CPUs using multi-queue NICs.

4.1 Principles and Practices

Our overall dataplane design philosophy is flexible, high-performance software coupled with hardware offloads. A high performance software dataplane can provide performance indistinguishable from the underlying hardware. We provision sufficient Fast Path CPU to achieve throughput targets, leveraging hardware offloads on a per-platform basis to minimize the Fast Path CPU required. Currently, we offload encryption; checksums; and memory copies using Intel QuickData DMA Engines [2]. We are investigating more substantial hardware offloads.

A software dataplane allows a uniform featureset and performance profile for customers running on heterogeneous hardware with different NICs and hardware offloads. This uniformity enables transparent live migration across heterogenous hardware. Whether a network feature has none, some, or all functionality in software or hardware becomes a per-platform detail. On-host software is extensible, supports rapid release velocity (see Section 6.2), and scales to large amounts of state. A full SR-IOV hardware approach requires dedicated middleboxes to handle new features or to scale beyond hardware table limits. Such middleboxes increase latency, cost, failure rates, and operational overhead.

To achieve high performance in software, the Fast Path design *minimizes Fast Path features*. Each feature we add to the Fast Path has a cost and consumes per-packet CPU budget. Only performance-critical low-latency work belongs on the Fast Path. Work that is CPU-intensive or does not have strict latency requirements, such as work specific to inter-cluster or Internet traffic, belongs on the Coprocessor Path. Our design also *minimizes per-flow work*. All VM packets go through the Fast Path routing flow table. We can optimize by using flow key fields to pre-compute per-flow work. For example, Andromeda computes an efficient per-flow firewall classifier during flow insertion, rather than requiring an expensive full firewall ruleset match for every packet.

The Fast Path uses high-performance best practices: avoid locks and costly synchronization, optimize memory locality, use hugepages, avoid thread handoffs, end-to-end batching, and avoid system calls. For example, the Fast Path only uses system calls for Coprocessor thread wakeups and virtual interrupts. The Fastpath uses lock-free Single Producer / Single Consumer (SPSC) packet rings and channels for communication with control and Coprocessor threads.

4.2 Fast Path Design

The Fast Path performs packet processing actions required for performance-critical VM flows, such as intra-cluster VM-VM. The Fast Path consists of separate ingress and egress engines for packet processing and other periodic work such as polling for commands from control threads. Engines consist of a set of reusable, connected packet processing push or pull *elements* inspired by Click [29]. Elements typically perform a single task and operate on a batch of packets (up to 128 packets on ingress and 32 on egress). Batching results in a 2.4x increase in peak packets per second. VM and Host NIC queues are the sources and sinks of element chains. To avoid thread handoffs and system calls, the Fast Path directly accesses both VM virtual NIC and host physical NIC queues via shared memory, bypassing the VMM and host OS.

Figures 6 and 7 outline the elements and queues for Andromeda Fast Path engines. The pull-pusher is the C++ call point into the element chain in both Fast Path engines. In the egress engine, the pull-pusher pulls packets from the VM transmit queue chain, and pushes the packets to the NIC transmit queue chain. In the ingress engine, the pull-pusher pulls packets from NIC receive queue chain and pushes the packets to the VM chain. Engine element chains must support fan-in and fan-out, as there may be multiple queues and VMs. Hash Demux elements use a packet 5-tuple hash to fan-out a batch of packets across a set of push elements. Schedulers pull batches of packets from a set of pull elements to provide fan-in. To scale to many VMs and queues per host, the VM Round Robin Scheduler element in the egress engine checks if a VM's queues are idle before calling the long chain of C++ element methods to actually pull packets from the VM. Routing is the core of the Fast Path, and is implemented by the ingress and egress flow table pipeline elements discussed in Section 4.3.

Several elements assist with debugging and monitoring. *Tcpdump* elements allow online packet dumps. *Stats exporter* records internal engine and packet metrics for performance tuning. *Packet tracer* sends metadata to an off-host service for network analysis and debug. *Latency sampler* records metadata for off-host analysis of network RTT, throughput, and other performance information.

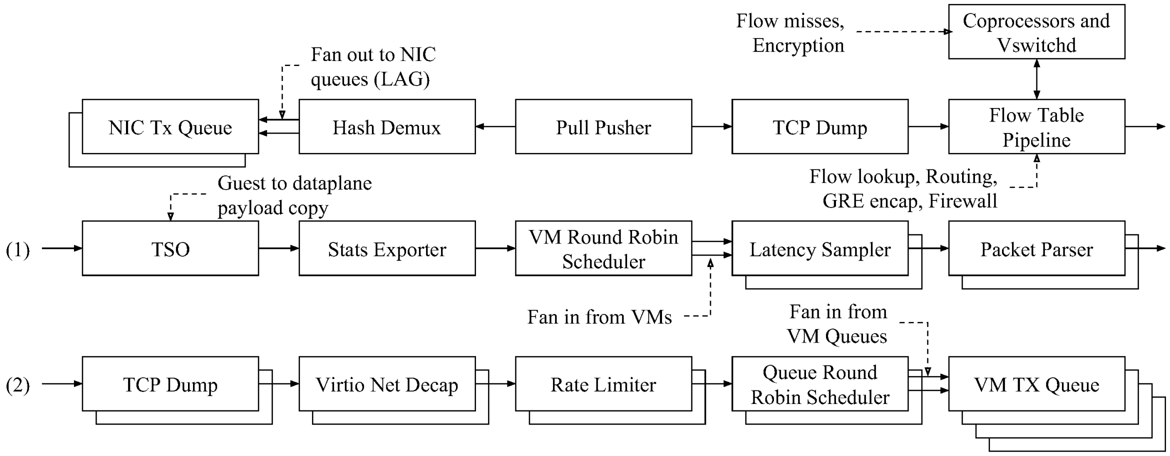


Figure 6: Andromeda dataplane egress engine. Note: Arrows indicate the direction of the push or pull C++ function call chain. Packets flow from the lower right to the upper left.

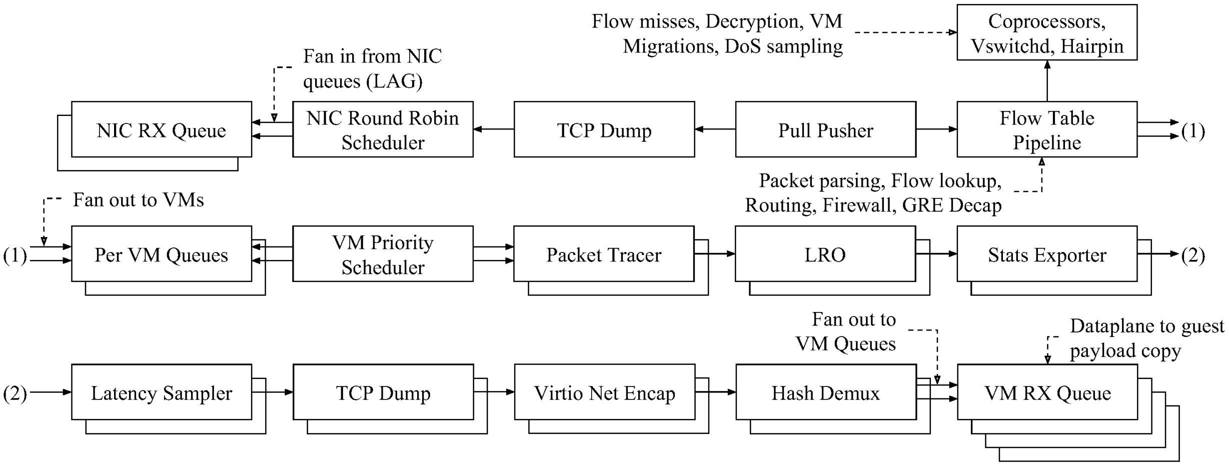

Flow misses, Decryption, VM Migrations, DoS sampling
Coprocessors, Vswitchd, Hairpin
Fan in from NIC queues (LAG)
NIC RX Queue
NIC Round Robin Scheduler
TCP Dump
Pull Pusher
Flow Table Pipeline
(1)
Packet parsing, Flow lookup, Routing, Firewall, GRE Decap
Fan out to VMs
(1)
Per VM Queues
VM Priority Scheduler
Packet Tracer
LRO
Stats Exporter
(2)
Fan out to VM Queues
Dataplane to guest payload copy
(2)
Latency Sampler
TCP Dump
Virtio Net Encap
Hash Demux
VM RX Queue

Figure 7: Andromeda dataplane ingress engine. Note: Arrows indicate the direction of the push or pull C++ function call chain. Packets flow from the upper left to the lower right.

4.3 Fast Path Flow Table

All VM packets pass through the engine flow table (FT) for both routing and per-flow packet operations such as encapsulation. The FT is a cache of the vswitchd flow tables. We *minimize per-flow work* through the FT in multiple ways. The FT uses only a single hash table lookup to map the flow key to a dense integer flow index. Subsequent tables, such as the action table and firewall policy table, are flat arrays indexed by flow index. Flow actions perform common packet operations such as encapsulation and setting the Ethernet header. Actions also store the destination virtual switch port and the set of Fast Path and Coprocessor packet stages for the flow, if any. Commonly, VM-VM intra-cluster flows have no packet stages enabled, and complete FT execution after applying the action set to the packet.

To avoid costly synchronization, the FT does not use locks, and is never modified by the engines. To update the FT, a control thread updates a shadow FT and then updates the engine via SPSC channels to point to the new FT [36]. Each engine maintains its own FT flow statistics, which vswitchd periodically reads and aggregates.

When a packet is sent through the FT, the FT computes the flow key for the packet, looks up the key in the flow index table, then applies the specified flow actions and any enabled Fast Path packet stages, and finally updates statistics. If Coprocessor stages are enabled for the flow, the packet is sent to the appropriate Coprocessor thread.

FT keys are either 3-tuple or 5-tuple. Ingress FT keys support encapsulation, and include both the inner and outer packet 3-tuple. Egress FT keys are unencapsulated. Flow index table lookup is attempted first with a 3-tuple flow key. If no match is found, a 5-tuple flow key is computed and lookup is retried. If no match is found again the packet is sent to the Flow Miss Coprocessor, which sends the packet to vswitchd. 3-tuple keys are used wherever possible, and are the common case for VM-VM traffic. Example uses of 5-tuple keys include VM connections to load balanced VIPs, as VIP backend selection is performed on a per-connection basis.

4.3.1 Middlebox Functionality

Andromeda provides middlebox functions such as firewall, load balancing, and NAT on-host [7, 27, 32]. This approach achieves higher performance and reduced provisioning complexity compared to traditional dedicated appliance middleboxes. A key challenge is how to ensure that these features do not degrade Fast Path performance. To accomplish this goal, we perform the expensive middlebox work in the on-host control plane on a flow miss. The control plane inserts the flow into the FT with any middlebox packet stage work pre-computed per-flow.

An example Fast Path feature is the always-on connection tracking firewall [4]. Traditional firewalls require a *firewall rule* lookup and a *connection-tracking table lookup* per packet, both of which are expensive. To *minimize per-flow work*, vswitchd analyzes the rules on a flow miss in order to minimize the amount of work that the Fast Path must do. If the IP addresses and protocol in the flow are always allowed in both directions, then no firewall work is needed in the Fast Path. Otherwise, vswitchd enables the firewall stage and computes a *flow firewall policy*, which indicates which port ranges are allowed for the flow IPs. The Fast Path matches packets against these port ranges, which is much faster than evaluating the full firewall policy. Furthermore, the firewall stage skips connection tracking if a packet connection 5-tuple is permitted by the firewall rules in both directions, which is a common case for VM-VM and server flows.

4.4 Coprocessor Path

The Coprocessor Path implements features that are CPU-intensive or do not have strict latency requirements. Coprocessors play a key role in *minimizing Fast Path features*, and decouple feature growth from Fast Path performance. Developing Coprocessor features is easier because Coprocessors do not have to follow the strict Fast Path best practices. For example, Coprocessor stages may acquire locks, allocate memory, and perform system calls.

Coprocessor stages include encryption, DoS, abuse detection, and WAN traffic shaping. The encryption stage provides transparent encryption of VM to VM traffic across clusters. The DoS and abuse detection stage enforces ACL policies for VM to internet traffic. The WAN traffic shaping stage enforces bandwidth sharing policies [26]. Coprocessor stages are executed in a per-VM floating Coprocessor thread, whose CPU time is attributed to the container of the corresponding VM. This design provides fairness and isolation between VMs, which is critical for more CPU-intensive packet work. A single coprocessor thread performing Internet ACL/shaping can send 11.8Gb/s in one netperf TCP stream, whereas one coprocessor thread performing WAN encryption without hardware offloads can send 4.6Gb/s. This inherent difference in per-packet processing overhead highlights the need for attributing and isolating packet processing work.

Fast Path FT lookup determines the Coprocessor stages enabled for a packet. If Coprocessor stages are enabled, the Fast Path sends the packet to the appropriate Coprocessor thread via an SPSC packet ring, waking the thread if necessary. The Coprocessor thread applies the Coprocessor stages enabled for the packet, and then returns the packet to the Fast Path via a packet ring.

5 Evaluation

This section evaluates the resource consumption, performance, and scalability of Andromeda.

5.1 On-Host Resource Consumption

Andromeda consumes a few percent of the CPU and memory on-host. One physical CPU core is reserved for the Andromeda dataplane. One logical CPU of the core executes the busy polling Fast Path. The other mostly idle logical CPU executes infrequent background work, leaving most of the core's shared resources available to the Fast Path logical CPU. In the future, we plan to increase the dataplane CPU reservation to two physical cores on newer hosts with faster physical NICs and more CPU cores in order to improve VM network throughput.

The Andromeda dataplane has a 1GB max memory usage target. To support non-disruptive upgrades and a separate dataplane lifecycle management daemon, the total dataplane memory container limit is 2.5GB. The combined vswitchd and host agent memory limit is 1.5GB.

5.2 Dataplane Performance

Dataplane performance has improved significantly throughout the evolution of Andromeda.

Pre-Andromeda was implemented entirely in the VMM and used UDP sockets for packet I/O. The dataplane supported only a single queue virtual NIC with zero offloads, such as LRO and TSO.

Andromeda 1.0 included an optimized VMM packet pipeline and a modified kernel Open vSwitch (OVS). We added virtual NIC multi-queue and egress offloads.

Andromeda 1.5 added ingress virtual NIC offloads and further optimized the packet pipeline by coalescing redundant lookups in the VMM with the kernel OVS flow table lookup. Host kernel scheduling and C-State management were also optimized, improving latency.

Andromeda 2.0 consolidated prior VMM and host kernel packet processing into a new OS-bypass busy-polling userspace dataplane. The VMM continued to handle VM virtual NIC ring access and interrupts, but all packet processing was performed by the new dataplane. The VMM exchanged packets with the dataplane via SPSC shared memory rings. The dataplane maps in all VM memory, and directly copies packet to/from VM memory.

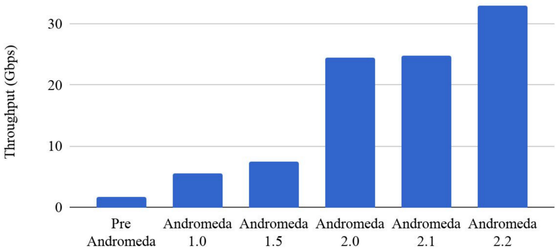


Figure 8: Multi-stream TCP throughput.

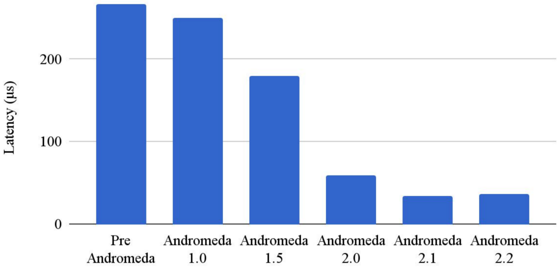


Figure 9: TCP round trip latency.

Andromeda 2.1 directly accesses VM virtual NIC rings in the dataplane, bypassing the VMM. Performance-critical packets are processed end-to-end by the busy-polling dataplane CPU without thread handoffs or context switches. Eliminating the VMM from the packet datapath removed four thread wakeups per network round trip which significantly improved latency and CPU efficiency.

Andromeda 2.2 uses Intel QuickData DMA Engines [2] to offload larger packet copies, improving throughput. DMA engines use an IOMMU for safety and are directly accessed by the dataplane via OS bypass. Tracking async packet copies and maintaining order caused a slight latency increase over Andromeda 2.1 for small packets.

Throughout this evolution, we tracked many performance metrics. Figure 8 plots the same-cluster throughput achievable by two VMs using 200 TCP streams. As the stack evolved, we improved throughput by 19x and improved same-cluster TCP round trip latency by 7x (Figure 9). Figure 10 plots the virtual network CPU efficiency in cycles per byte during a multi-stream benchmark. Our measurement includes CPU usage by the sender and receiver for both the VM and host network dataplane. The host dataplane covers all host network processing, including the VMM, host kernel, and new Andromeda OS bypass dataplane. Overall, Andromeda reduced cycles per byte by a factor of 16. For Andromeda 2.0 and later, we use the host resource limits described in Section 5.1 and execute the Fast Path on a single reserved logical CPU.

We measured these results on unthrottled VMs connected to the same Top of Rack switch. Benchmark hosts have Intel Sandy Bridge CPUs and 40Gb/s physical NICs except for Pre-Andromeda, which used bonded 2x10Gb/s NICs. The sender and receiver VMs run a Linux 3.18

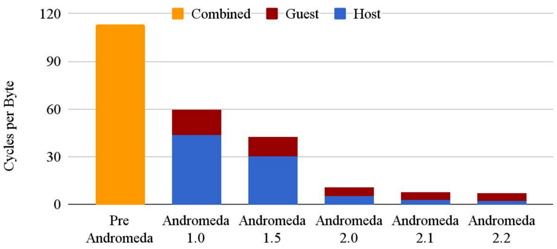


Figure 10: CPU Efficiency. Note: Guest versus host breakdown is unavailable for Pre-Andromeda

guest kernel, and are configured with 8 VCPUs. The only hardware offloads used are checksum offloads, and in Andromeda 2.2, memory copy offloads.

5.3 Control Plane Agility and Scale

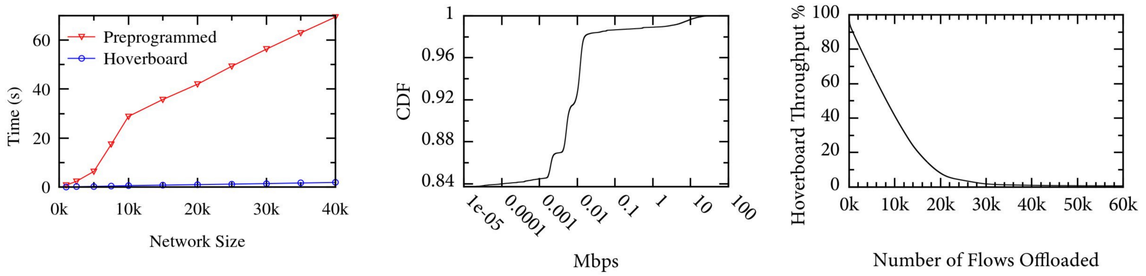
Andromeda 1.0 used the preprogrammed model for all VM-VM flows, and initially supported networks up to 2k VMs. Under the preprogrammed model, even a small change in a virtual network, such as adding a VM, requires updating the routing tables for all other VMs in the network.

We subsequently developed the Hoverboard model, which made the control plane substantially more scalable and agile. With Hoverboards, a new VM has network connectivity as soon as its host and the Hoverboards are programmed. Median programming latency – the time required for the VM controller to process an FM request (e.g., to add a VM to a network) and program the appropriate flow rules via OpenFlow – improved from 551ms to 184ms. The 99th percentile latency dropped even more substantially, from 3.7s to 576ms. Furthermore, the control plane now scales gracefully to virtual networks with 100k VMs.

Figure 11a shows VMC flow programming time for networks with varying numbers of VMs. The control plane has 30 VMC partitions with 8 Broadwell logical CPUs per partition, and 60 OFEs with 4 CPUs each. VMs are scheduled on a fixed pool of 10k physical hosts.

When using the Hoverboard model, the programming time and number of flows grows linearly in network size. However, when we use the preprogrammed model, the programming time and number of flows is $O(n \times h)$, where n is the number of VMs and h is the number of hosts with at least one VM in the network. Multiple VMs on the same host can share a forwarding table; without this optimization, the number of flows would be $O(n^2)$. Programming time grows quadratically in n until $n \approx h$, and linearly thereafter. Quadratic growth is representative of typical multi-regional deployments, for which $n \ll h$.

For the 40k VM network, VMCs program a total of 1.5M flows in 1.9 seconds with the Hoverboard model, using a peak of 513MB of RAM per partition. Under the preprogrammed model, VMCs program 487M flows



(a) Programming time vs. network size **(b)** Bandwidth distribution across flows **(c)** Hoverboard throughput vs. offloaded flows

Figure 11: Hoverboard scaling analysis

in 74 seconds, and peak RAM use per partition is 10GB. Beyond 40k VMs, preprogramming led to stability issues in the OFE and vswitchd due to the large number of flows.

The effectiveness of the Hoverboard model is predicated on the assumption that a small number of offloaded flows installed by the control plane can capture most of the traffic, with a long tail of small flows going through the Hoverboards. Indeed, network flows in our clusters follow a power law distribution, consistent with prior observations, e.g., [19]. Figure 11b shows a CDF of peak throughput across all VM pairs in a production cluster, measured over a 30-minute window. 84% of VM pairs never communicate: in the Hoverboard model, these flows are never programmed, and their host overhead is zero. 98% of flows have peak throughput less than 20kbps, so with an offload threshold of 20kbps, Hoverboards improve control plane scalability by 50x.

Figure 11c shows the peak traffic through the Hoverboard gateways as we program more direct host to host flows. The figure shows that by shifting a total of about 50k flows (less than 0.1% of the total flows possible) to end hosts, the peak throughput through the Hoverboard gateways drops to less than 1% of the cluster bandwidth.

6 Experiences

This section describes our experiences building Andromeda, challenges we faced as the system grew and evolved, and how we addressed those challenges.

6.1 Resource Management

In a Cloud environment, it is essential to provide isolation among tenants while making effective use of resources. Here we discuss CPU and memory; for a description of how we manage network bandwidth, see [26].

6.1.1 CPU Isolation

Andromeda 1.0 shipped with a kernel datapath that provided limited isolation. Ingress traffic was processed on kernel softirq threads shared by all VMs on-host. These softirq threads could run on any host CPU and did not have CPU attribution. The current Andromeda userspace datapath performs Fast Path processing on its own dedicated CPU. Packets requiring CPU-intensive processing are sent to a per-VM coprocessor thread which is CPU attributed to the VM container

As the Fast Path is shared by all VMs on-host, Andromeda provides isolation within the Fast Path. For egress, the Fast Path polls VMs round robin and each VM is rate limited to a VM virtual NIC line rate. Initially, we provided no Fast Path isolation between VMs for ingress. This resulted in noisy neighbor problems when the packet rate of one VM was higher than the Fast Path could pull packets off the physical NIC. The NIC queue would back up and drop packets, harming other VMs on-host.

To improve isolation, we split the Fast Path ingress engine into two parts. The front half polls the NIC, performs flow table lookup, and places packets into per-VM queues. The more CPU intensive back half pulls packets from the per-VM queues, copies the packets to the VM, and may raise a virtual interrupt. In Figure 7, the back half begins after the Per VM Queues stage. Isolation is provided within the back half by the VM Priority Scheduler. The scheduler selects the per-VM queue whose recent back half processing has consumed the least Fast Path CPU. Unfortunately, noisy neighbor issues may still arise if the less CPU intensive front half is overloaded. In the future we will explore use of per-VM physical NIC queues to provide isolation in the front half as well.

6.1.2 Guest VM Memory

The Andromeda dataplane maps in all of guest VM memory for all VMs on-host, as any guest memory address may be used for packet I/O. However, dataplane access to VM memory creates attribution and robustness challenges. Guest VM memory is backed by a host tmpfs file and is demand-allocated. Dataplane access to an unbacked VM memory page causes a page allocation, which is typically charged to the process triggering the allocation. To ensure that the dataplane does not exceed its memory limits (Section 5.1), we modified our host Linux kernel to always charge VM memory page allocations to the VM memory container rather than the process triggering the allocation.

The dataplane memory container is also charged for kernel page table memory. Over time, the dataplane can access hundreds of GB of VM memory. If page table memory usage exceeds a target limit, Andromeda asks the kernel to free page table entries. This is done by mmaping the VM memory file over its existing mapping, which atomically replaces the existing mapping. Remapping a guest VM memory region is done in multiple mmap calls over small chunks rather than one system call. This avoids tail latency due to kernel memory lock contention.

A VM can crash while the dataplane is accessing VM memory. A compromised VMM could also truncate the VM memory backing file. To handle these cases, all dataplane VM memory access occurs via custom assembly memcpy functions. On VM memory access failure, the dataplane receives a signal. The dataplane signal handler modifies the signal saved registers so that execution resumes at a custom memcpy fixup handler, which returns failure to the memcpy caller. On memcpy failure, the dataplane disconnects the offending VM's queues. We must minimize the Fast Path cost relative to normal memcpy in the common case of VM memory access success. Our approach requires only a single extra branch to test the memcpy return value. This is similar to how the Linux kernel handles failure during memory copies to userspace.

6.1.3 Memory Provisioning

Our Cloud environment serves a diverse set of applications which place varying memory demands on the network stack due to routing tables, firewall connection tracking, load balancing, etc. Therefore, a key challenge is how to provision memory with minimal waste.

We initially expected network virtualization features to span the Top of Rack switch (ToR), switch fabric, and host machine software. The fact that we already managed ToR and switch fabrics with OpenFlow contributed to our decision to use OpenFlow for Andromeda. However, we found that scaling a feature to many virtual networks is much easier on an end host where we can provision additional memory and CPU per network.

Currently we statically provision dataplane memory on each host (Section 5.1), regardless of the number of VMs running on the host or how those VMs use networking features. We are exploring attribution of dataplane networking memory usage to VM containers so that network features may dynamically scale memory usage. This approach reduces dataplane memory overprovisioning and allows the cluster manager to take network memory usage for a VM into account during VM placement.

6.2 Velocity

The Cloud ecosystem is evolving rapidly, so a key challenge was to build a platform with high feature velocity. Our strategy for achieving velocity has evolved over time.

6.2.1 Andromeda 1.0: Kernel Datapath

Andromeda 1.0 shipped with the Open vSwitch kernel datapath. Kernel upgrades were much slower than our control plane release cycle. The OpenFlow APIs provide a flexible flow programming model that allowed us to deploy a number of features with only control plane changes. OpenFlow was instrumental in getting the project off the ground and delivering some of the early features, such as load balancing and NAT, via control plane changes alone.

However, we also faced a number of difficulties. For example, OpenFlow was not designed to support stateful features such as connection tracking firewalls and load balancing. We initially tried to express firewall rules in OpenFlow. However, expressing firewall semantics in terms of generic primitives was complex, and the implementation was difficult to optimize and scale. We ultimately added an extension framework in OVS to support these features. Unlike OpenState [8] and VFP [13], which integrate stateful primitives into the flow lookup model, our extensions use a separate configuration push mechanism.

6.2.2 Andromeda 2.0: Userspace Datapath

Qualifying and deploying a new kernel to a large fleet of machines is intrinsically a slow process. Andromeda 2.0 replaced the kernel dataplane with a userspace OS bypass dataplane, which enabled weekly dataplane upgrades. At that point, the advantage of having a programming abstraction as generic as OpenFlow diminished.

A rapid dataplane release cycle requires non-disruptive updates and robustness to rare but inevitable dataplane bugs. To provide non-disruptive updates, we use an upgrade protocol consisting of a brownout and blackout phase. During brownout, the new dataplane binary starts and transfers state from the old dataplane. The old dataplane continues serving during brownout. After the initial state transfer completes, the old dataplane stops serving and blackout begins. The old dataplane then transfers delta state to the new dataplane, and the new dataplane begins serving. Blackout ends for a VM when the VMM connects to the new dataplane and attaches its VM virtual NIC. The median blackout time is currently 270ms. We plan to reduce blackout duration by passing VMM connection file descriptors as part of state transfer to avoid VMM reconnect time.

The userspace dataplane improves robustness to dataplane bugs. The VM and host network can be hardened so that a userspace dataplane crash only results in a temporary virtual network outage for the on-host VMs. On-host control services continuously monitor the health of the dataplane, restarting the dataplane if health checks fail. Host kernel networking uses separate queues in the NIC, so we can roll back dataplane releases even if the dataplane is down. In contrast, a failure in the Linux kernel network stack typically takes down the entire host.

6.3 Scaling and Agility

Our initial pre-Andromeda virtual network used a global control plane, which pre-programmed routes for VM-VM traffic, but also supported on-demand lookups to reduce perceived programming latency. The on-demand model was not robust under load. When the control plane fell behind, all VM hosts would request on-demand programming, further increasing the load on the control plane, leading to an outage.

Andromeda initially used OpenFlow and the pre-programmed model exclusively. As we scaled to large networks, we ran into OpenFlow limitations. For example, supporting a million-entry IP forwarding table across a large number of hosts requires the control plane to transmit a compact representation of the routes to the dataplane. Expressing such a large number of routes in OpenFlow adds unacceptable overhead. As another example, the Reverse Path Forwarding check in OpenFlow required us to duplicate data that was already present in the forwarding table, and we built a special-purpose extension to avoid this overhead. We also scaled the control plane by partitioning VM Controllers and parallelizing flow generation. However, the pre-programmed model's quadratic scaling (see Section 5.3) continued to create provisioning challenges, particularly for on-demand customer workloads.

Later, we introduced the Hoverboard model, which lead to more stable control plane overhead and enabled much faster provisioning of networks (see Section 5.3). Hoverboards have allowed us to scale to support large virtual networks. However, certain workloads create challenges. For example, batch workloads in which many VM pairs begin communicating simultaneously at high bandwidth consume substantial Hoverboard capacity until the control plane can react to offload the flows. We have made a number of improvements to address that, such as detecting and offloading high-bandwidth flows faster.

7 Related Work

The Andromeda control plane builds upon a wide body of software defined networking research [10, 15, 25], OpenFlow [3, 28] and Open vSwitch [33]. The data plane design overlaps with concepts described in Click [29], SoftNIC [17], and DPDK [1].

NVP [24] is a SDN-based network virtualization stack, like Andromeda. NVP also uses Open vSwitch, along with the OVS kernel datapath, similar to Andromeda 1.0. The NVP control plane uses the preprogrammed model (Section 3.3), so a network with n VMs will have $O(n^2)$ flows. NVP scales by using partitioning, and by dividing the control plane into virtual and physical layers. Andromeda also uses partitioning, but principally solves the scaling issue by using the Hoverboard model.

VFP [13] is the SDN-based virtualization host stack for Microsoft Azure, using a layered match-action table model with stateful rules. All VFP features support offloading to an exact-match fastpath implemented in the host kernel or an SR-IOV NIC. Andromeda uses a hierarchy of flexible software-based userspace packet processing paths. Relative to VFP, Coprocessors enable rapid iteration on features that are CPU-intensive or do not have strict latency targets, and allow these features to be built with high performance outside of the Fast Path. Andromeda does not rely on offloading entire flows to hardware: We demonstrate that a flexible software pipeline can achieve performance competitive with hardware. Our Fast Path supports 3-tuple flow lookups and minimizes use of stateful features such as firewall connection tracking. While the VFP paper does not focus on the control plane, we present our experiences and approach to building a highly scalable, agile, and reliable network control plane.

8 Conclusions

This paper presents the design principles and deployment experience with Andromeda, Google Cloud Platform's network virtualization stack. We show that an OS bypass software datapath provides performance competitive with hardware, achieving 32.8Gb/s using a single core. To achieve isolation and decouple feature growth from fastpath performance, we execute CPU-intensive packet work on per-VM Coprocessor threads. The Andromeda Control plane is designed for agility, availability, isolation, feature velocity, and scalability. We introduce Hoverboards, which makes it possible to program connectivity for tens of thousands of VMs in seconds. Finally, we describe our experiences deploying Andromeda, and we explain how non-disruptive upgrades and VM live migration were critical to navigating major architectural shifts in production with essentially no customer impact.

In the future, we will offload more substantial portions of the Fast Path to hardware. We will continue to improve scalability, as lightweight virtual network endpoints such as containers will result in much larger and more dynamic virtual networks.

Acknowledgements

We would like to thank our reviewers, shepherd Michael Kaminsky, Jeff Mogul, Rama Govindaraju, Aspi Siganporia, and Parthasarathy Ranganathan for paper feedback and guidance. We also thank the following individuals, who were instrumental in the success of the project: Aspi Siganporia, Alok Kumar, Andres Lagar-Cavilla, Bill Sommerfeld, Carlo Contavalli, Frank Swiderski, Jerry Chu, Mike Bennett, Mike Ryan, Pan Shi, Phillip Wells, Phong Chuong, Prashant Chandra, Rajiv Ranjan, Rüdiger Sonderfeld, Rudo Thomas, Shay Raz, Siva Sunkavalli, and Yong Ni.

References

[1] Intel data plane development kit. http://www.intel.com/go/dpdk.

[2] Intel I/O acceleration technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html.

[3] Openflow specification. https://www.opennetworking.org/software-defined-standards/specifications/.

[4] Using networks and firewalls. https://cloud.google.com/compute/docs/networking.

[5] M. Baker-Harvey. Google compute engine uses live migration technology to service infrastructure without application downtime. https://cloudplatform.googleblog.com/2015/03/Google-Compute-Engine-uses-Live-Migration-technology-to-service-infrastructure-without-application-downtime.html.

[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[7] S. M. Bellovin. Distributed firewalls. *IEEE Communications Magazine*, 32:50–57, 1999.

[8] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM*, 44(2):44–51, Apr. 2014.

[9] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[10] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 1–12, New York, NY, USA, 2007. ACM.

[11] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[12] D. E. Eisenbud, C. Yi, C. Contavalli, C. Smith, R. Kononov, E. Mann-Hielscher, A. Cilingiroglu, B. Cheyney, W. Shang, and J. D. Hosein. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, pages 523–535, Berkeley, CA, USA, 2016. USENIX Association.

[13] D. Firestone. VFP: A virtual switch platform for host SDN in the public cloud. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 315–328, Boston, MA, 2017. USENIX Association.

[14] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. OpenNF: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 163–174, New York, NY, USA, 2014. ACM.

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4D approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, Oct. 2005.

[16] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, Feb. 2015.

[17] S. Han, K. Jang, A. Panda, S. Palkar, D. Han, and S. Ratnasamy. SoftNIC: A software NIC to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[18] M. R. Hines, U. Deshpande, and K. Gopalan. Post-copy live migration of virtual machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[19] R. Jain. Characteristics of destination address locality in computer networks: A comparison of caching schemes. *Computer Networks and ISDN Systems*, 18:243–254, 1989.

[20] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[21] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella. Paving the way for NFV: Simplifying middlebox modifications using statealyzr. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 239–253, Santa Clara, CA, 2016. USENIX Association.

[22] C. Kim, M. Caesar, and J. Rexford. Floodless in Seattle: A scalable ethernet architecture for large enterprises. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 3–14, New York, NY, USA, 2008. ACM.

[23] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association.

[24] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip, and R. Zhang. Network virtualization in multi-tenant datacenters. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 203–216, Seattle, WA, 2014. USENIX Association.

[25] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Z. Google, R. Ramanathan, Y. I. NEC, H. I. NEC, T. H. NEC, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 351–364, Berkeley, CA, USA, 2010.

[26] A. Kumar, S. Jain, U. Naik, A. Raghuraman, N. Kasinadhuni, E. C. Zermeno, C. S. Gunn, J. Ai, B. Carlin, M. Amarandei-Stavila, M. Robin, A. Siganporia, S. Stuart, and A. Vahdat. BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 1–14, New York, NY, USA, 2015. ACM.

[27] J. Lee, J. Tourrilhes, P. Sharma, and S. Banerjee. No more middlebox: Integrate processing into network. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 459–460, New York, NY, USA, 2010. ACM.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[29] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 217–231, New York, NY, USA, 1999. ACM.

[30] M. Nelson, B.-H. Lim, and G. Hutchins. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference*, USENIX '05, pages 391–394, Berkeley, CA, USA, 2005. USENIX Association.

[31] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 39–50, New York, NY, USA, 2009. ACM.

[32] P. Patel, D. Bansal, L. Yuan, A. Murthy, A. Greenberg, D. A. Maltz, R. Kern, H. Kumar, M. Zikos, H. Wu, C. Kim, and N. Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[33] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, 2015. USENIX Association.

[34] L. Rizzo. Netmap: A novel framework for fast packet i/o. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[35] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: Live router migration as a network-management primitive. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 231–242, New York, NY, USA, 2008. ACM.

[36] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM.