Multitenancy for Fast and Programmable Networks in the Cloud

Tao Wang[†*] Hang Zhu[⋆*] Fabian Ruffy[†] Xin Jin[⋆] Anirudh Sivaraman[†]
Dan R. K. Ports[⋆⋆] Aurojit Panda[†]

[†]*New York University* [⋆]*Johns Hopkins University* [⋆⋆]*Microsoft Research*

Abstract

Fast and programmable network devices are now readily available, both in the form of programmable switches and smart network-interface cards. Going forward, we envision that these devices will be widely deployed in the networks of cloud providers (e.g., AWS, Azure, and GCP) and exposed as a programmable surface for cloud customers—similar to how cloud customers can today rent CPUs, GPUs, FPGAs, and ML accelerators. Making this vision a reality requires us to develop a mechanism to share the resources of a programmable network device across multiple cloud tenants. In other words, we need to provide *multitenancy* on these devices. In this position paper, we design compile and run-time approaches to multitenancy. We present preliminary results showing that our design provides both efficient resource utilization and isolation of tenant programs from each other.

1 Introduction

New programmable network devices can run custom logic at performance levels previously reserved for custom fixed-function hardware. Leveraging a large body of research on reconfigurable architectures [16, 18] and programming languages [43, 55], both programmable switches [14] and network interface cards (NICs) [4, 47] are now widely available. We envision a world where such programmable network devices could be widely deployed at scale within the networks of a cloud provider [11, 12, 23] and where their programmable surfaces could be exposed to cloud customers—similar to today's cloud compute [11, 21], storage [48], and networking [7]. To achieve this vision, we must answer a basic question: *how do we allow multiple programs from multiple tenants to coexist on a programmable network device?*

Today's programmable network devices do not support multitenancy. In effect, they exist in the 1950s world of pre-timesharing mainframes: a single user can load a single program at a time. Our aim is to provide the missing "operating system" that allows running multiple separate programs on a network device. Like a traditional OS, its goal is to efficiently share resources and to enforce isolation between programs. However, the traditional time-sharing approach is infeasible on high-speed network devices because they lack the hardware support to run an OS. New techniques are required.

This paper presents a vision for multitenancy on programmable network devices. It targets devices with programmable ASIC-based pipelines, e.g., the Barefoot Tofino switch architecture [14]. We argue that multitenancy is valuable for these devices (§2), and that a hybrid compile-time/run-time architecture serves as the right OS for multitenancy (§4). As evidence for our design's feasibility, we present a preliminary evaluation (§5). We show that multiple programs that are isolated from each other can still coexist to efficiently use resources on the Barefoot Tofino programmable switch.

2 Motivating Multitenancy

The advent of programmable networks has led to many proposals to use in-network computation to improve networked software and applications. Researchers have exploited programmable packet processing logic to develop and prototype new algorithms for classical network problems including congestion control [51, 52], packet scheduling [54], and load balancing [41]. It has also enabled new application-specific uses for in-network processing, including accelerators for consensus [19, 35], storage [27, 28], databases [33, 34, 59], and machine learning [49].

The diversity of these systems highlights an important point: different applications and different tenants have different needs. Unfortunately, the lack of multitenancy support forces network operators to either run a single program on a given device, or to integrate multiple in-network applications (along with the baseline switch functionality) into a monolithic program, a painstaking and error-prone process.

Adding multitenancy support to programmable networks offers cloud network operators and users several advantages:

[*]Equal contribution.

Name	Description
Exact match crossbar	Extract packet bits for exact matching
Hash unit	Hash bits for exact match keys
Ternary match crossbar	Extract packet bits for ternary matching
SRAMs	Storage for exact match-action entry
TCAMs	Storage for ternary match-action entry
Action units	Perform operations on headers
Gateway tables	Perform conditional check
PHV containers	Store packet headers
Stateful Memory	Register, counter, meter, etc.

Table 1: Typical hardware resources in a pipeline

Enabling innovation. For datacenter network operators, network reliability is by far the highest priority [25]. Accordingly, they are loath to deploy new in-network applications that might interfere with basic network functionality. Providing isolation between basic routing/forwarding functionality and new programs—particularly if coupled with formal non-interference guarantees—addresses this concern.

Improving resource utilization. Few applications require the full computation and memory resources of a single device. As our preliminary analysis (§5) shows, most applications require a fraction of the different computational and memory resources on a programmable packet-processing pipeline. Multiplexing them onto a single device makes more efficient use of the pipeline's resources.

Enabling new cloud offerings. Today, programmable networking functionality is used by cloud providers to support their own operations. However, new cloud products offer access to accelerators like FPGAs [10], and customer-programmable networks are a logical next step. Making this vision a reality requires multitenancy with strong isolation.

3 Hardware Background

We are focused specifically on multitenancy for switches and NICs with a hardware architecture based on programmable packet-processing pipeline ASICs [4, 14, 16, 47]. We focus on this category of devices vs. other architectures such as multi-core processors [37, 40, 56] or FPGAs [22] for two reasons. First, some of these other architectures already provide some support for multitenancy in the form of OSes [38] or FPGA-based partial reconfiguration [9, 30]. Second, an ASIC architecture can achieve the highest per-device throughput. In particular, as cloud providers look ahead to a Terabit Ethernet era [5], we expect ASIC-based pipelines—both for switches and NICs—to be more effective than other programmable architectures in terms of both cost and power consumption.

Although different programmable packet-processing pipelines have different capabilities, they share the same conceptual blocks [53]: a configurable parser, one or more programmable pipelines (generally an ingress and egress pipeline), and queues for packet scheduling. The behavior of the parser and processing pipelines is specified in a language like P4 [43] and compiled into an ASIC configuration.

Parser. The parser identifies packet headers using a format specified by the P4 program. For example, a typical parser format will have an Ethernet header followed by an IP header and a TCP header. The parser stores a packet's extracted header data in a structure called a *packet header vector* (PHV). PHVs are streamed from one pipeline stage to the next for programmable packet processing.

Processing pipeline. The parsed PHVs together with per-packet metadata are passed through pipelines consisting of multiple *match-action stages*. Metadata can be program-defined (e.g., for temporary variables) or hardware-specific (e.g., queue size seen by the packet [2]). Each stage is associated with tables of rules. Part of the PHV is matched against these tables—using either an exact or ternary (wildcard) match—to identify a corresponding action. A simple example would be to match on a MAC address and select an output port. More complex actions use a collection of parallel action units in each stage to perform computations on the PHV. Action units can also access and modify persistent state of different types, e.g., registers, counters, or meters.

Resources. Pipelines have a fixed length. Programs either run at full line rate, or do not fit [53]. The compiler allocates various types of hardware resources, shown in Table 1: different programs demand different types. For example, a program with large exact or ternary match tables may be limited by the SRAM or TCAM required to store the tables.

4 Mechanisms for Multitenancy

Multitenancy imposes two basic requirements:

1. **Resource efficiency:** The resources of a programmable device should be efficiently shared across multiple different tenants. This involves consolidating functionality common to all tenants (e.g., packet forwarding) and ensuring there is modest overhead from running multiple tenant programs simultaneously on the same device.
2. **Isolation:** A tenant should be able to access only the resources allocated to it. It should not be able to interfere with the packet processing of a different tenant.

How should these requirements be enforced? Classically, this would be done by a time-sharing OS, monitor, or hypervisor. However, this approach is infeasible for the network devices we consider, which lack hardware support for isolation or context switching. Although hardware will evolve, we believe this limitation is fundamental given the strict timing constraints involved in packet-processing ASICs. We take a space-sharing approach instead, where resources are partitioned among programs.

Our design for multitenancy uses both compile-time and run-time components. The compile-time component is a linker that takes as inputs multiple tenant programs and merges them together, along with a system-level program

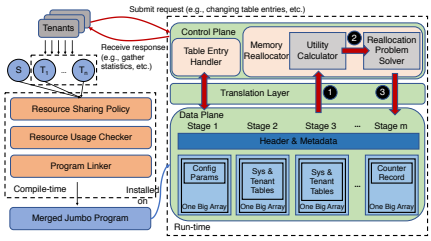

Figure 1: Multitenancy design. S: system, T: tenant.

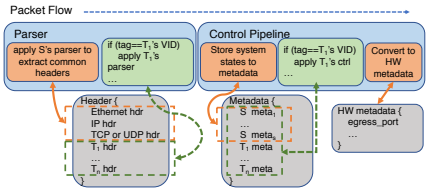

Figure 2: Program linker module.

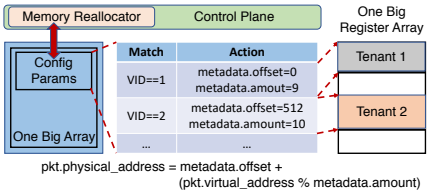

Figure 3: Page table in first stage.

that provides common packet processing functionality, into a single program that can be fed to the compiler of the network device. While doing so, the linker ensures that each tenant stays within its assigned resource allocation and that tenants do not interfere with each other.

One might think that this compile-time linker is sufficient by itself. However, it is a static approach. Changing the resource allocation requires re-merging, re-compiling, and reloading the resulting binary. While this is sufficient for adding and removing tenants, memory allocation requires a more dynamic approach because a tenant's memory usage can change depending on the workload. We address this with a virtual memory approach that uses run-time mechanisms to allocate and reclaim stateful memory from tenants.

The static and dynamic approaches used here complement each other well: the linker provides safe and efficient sharing for most resources, and the memory allocator provides dynamic resource allocation for one specialized resource (stateful memory). Ideally, a future system might be able to take a more dynamic approach to allocating all switch resources. Such a mechanism could be aided by hardware support for partial reconfiguration [9, 30]: the ability to partially update the switch/NIC binary without having to update it entirely.

4.1 System-Level Program

We assume tenants operate in a shared environment using virtual IP addresses (e.g., virtual private clouds [1, 7, 8]). These virtual IP addresses are local to each tenant and are meaningful only within the context of a tenant's own VMs and switch/NIC programs. To differentiate each tenant's traffic, we use a lightweight packet tagging method. Specifically, we use the VLAN ID (VID) within the 802.1Q header as a tenant identifier. Each tenant writes a packet-processing program that specifies a tenant's own functionality (e.g., NetCache [28]).

The system-level data-plane program provides functionality that is common to all tenants to avoid duplication across multiple tenant programs, which improves resource efficiency of the underlying network device. This program provides a parser for standard packet formats, currently Ethernet followed by IPv4 and TCP or UDP; tenant programs then provide tenant-specific parsers for the rest of the packet. This program also provides functionality for translating virtual IP addresses to physical IP addresses and MAC-address and IP-address-based forwarding. It maintains forwarding tables for the physical network on top of which the tenants are running. In addition, it measures global statistics (e.g., link utilization) that can be read, but not written, by tenant programs.

A system-level controller in the control plane controls the system-level data-plane program. This controller includes the run-time memory reallocator (§4.3). It also maintains the system-level program's tables (e.g., the physical network's routing tables), and modifies entries of the tenants' tables after ensuring that a tenant is only modifying its own tables.

4.2 Compile-Time Linker

Our compile-time linker consists of 3 modules (Figure 1): the resource sharing policy, the resource usage checker, and the program linker. The resource sharing policy specifies how to allocate resources among tenants (e.g., dominant resource fairness (DRF) [24], per-tenant quotas, or pay-as-you-go resource pricing). The resource usage checker checks that each tenant's resource usage is below the tenant's allocation. The program linker merges tenant-specific parsers and packet-processing programs with the system-level parser and program.

For the ingress and egress pipeline logic, the program linker uses a "sandwich" architecture when merging tenant programs with the system program as shown in Figure 2. Because the pipeline has a feed-forward architecture, packets can only flow from one stage to the next but not backward. Thus any communication between the tenant program and system program must be consistent with the pipeline's feed-forward nature. Hence, we allow the tenant program to *read* from the system program in the first stage (e.g., to read system-level statistics such as link utilization into metadata) and *write* to the system program in the last stage (e.g., to specify a virtual address for the system program to turn into a physical address). Hence, tenant programs are effectively sandwiched between the read and write halves of the system program.

For isolation between tenants, we mangle/rename all tenant programs' fields (e.g., tenant-defined headers and table names) to unique names to limit programs to their own namespaces. This allows two tenants to modify a packet header with the same name without interfering with each other. A single tenant can deploy their program on multiple programmable devices [27, 35] spanning the network path between a tenant's VMs. To ensure a tenant's program only processes packets belonging to that tenant in every device, we ensure that tenant programs can not modify the packet tag (i.e., VID). Permitting

tag modifications could cause a different tenant's program to process packets with a modified tag at downstream devices.

Isolation can be harmed by malicious tenants. For example, if one tenant has an 80-byte tenant-defined header, but only sends a packet with a 50-byte header, the remaining 30 bytes of the tenant header could be filled in with the corresponding bytes of a previously parsed packet from a different tenant. To prevent such side channels resulting from cached data from a previous packet, the merged program zeroes out all PHV containers before executing tenant logic. While programmable pipelines can ensure line-rate processing, malicious programs can recirculate packets into the pipeline or loop back packets through multiple devices, reducing pipeline resources for other tenants. To exclude this, we forbid recirculation in tenant programs and enforce loop-free routes in the control plane.

After carrying out the checks above, the linker merges the tenant programs into a single monolithic jumbo program. As an optimization, overlapping components (e.g., the parser) can be identified and consolidated among tenants programs during merging to conserve hardware resources [58]. We leave such consolidation to future work. The tenant and system programs are all written in P4 and hence the jumbo program is also in P4. This jumbo program is then fed to the device compiler and installed on the device. Each tenant has a controller in the control plane that can interact with the tenant's tables in the data plane by passing appropriate commands to the system-level controller, which executes commands via a translation layer on the tenants' behalf. The translation layer is responsible for converting virtual IPs to physical IPs.

4.3 Run-Time Memory Allocator

On-chip stateful memory is limited (tens of MB) and critical to many in-network applications [27, 28, 31, 33–36, 39, 49, 59]. However, traffic from these applications is variable [15]. Hence, it is advantageous to dynamically reallocate stateful memory to different tenants running such memory-intensive applications as their traffic demands change. Hence, we propose a virtual memory abstraction to dynamically allocate stateful memory to tenants at runtime. Such dynamic resource allocation on programmable packet-processing pipelines is nontrivial because resource usage is typically decided and compiled into the binary at compile time.

Typically, to use stateful memory, a programmer has to declare different P4 register arrays and hardcode their sizes (i.e., array width and height) at compile time. These sizing decisions might be incorrect for the traffic the tenant's program actually sees at run time. They are also inflexible, preventing us from taking away memory from one tenant and allocating it to another depending on each tenant's traffic patterns. To address this issue, we provide a novel *virtual stateful memory abstraction* which consists of three components, as shown in the data-plane of Figure 1: (i) one big register array in each pipeline stage that contains all the stateful memory of the stage; (ii) an initial configuration stage where the memory boundary of each tenant's current memory allocation is stored—effectively a page table for the one big register arrays; and (iii) the memory reallocator located in the control plane.

For each incoming packet from a tenant, the tenant's allocated memory parameters (e.g., offset and amount of allocated memory) are loaded into metadata in the first stage. These parameters are used by the tenant's packet to index the one big register arrays in subsequent stages. Thus the first stage serves as a page table for virtual memory in subsequent stages (Figure 3). To reallocate stateful memory dynamically, the memory reallocator changes each tenant's memory offsets and amounts based on an allocation policy and updates these quantities in the first stage. Additionally, the memory reallocator zeroes out newly reallocated memory regions to disallow tenants from accessing stale data from other tenants.

Resource efficiency. Tenants typically adjust memory usage dynamically for network measurement (e.g., [29, 42]). Our approach orchestrates such measurements by reallocating the stateful memory based on the utility of the memory to the different tenants. Figure 1 shows the main workflow of the dynamic resource allocation. The memory reallocator running in the control plane pulls the statistics (e.g., the request counter and the cache miss counter for NetCache [28]) from the data-plane periodically ❶. Then the utility estimator estimates the current utility for every tenant ❷. The reallocation problem solver takes the current utilities of tenants as input and generates a new resource allocation ❸.

Isolation. It is desirable to guarantee that every memory reallocation at run-time satisfies a formal high-level non-interference specification (e.g., any slot in the one big array is owned by at most one tenant). To this end, we design a set of verifiable API functions for memory reallocation. We assume that primitive control-to-data-plane operations [14, 45] are correct. Then, we prove that the memory reallocation implementation in terms of these primitives satisfies non-interference properties. We do so by translating both the properties and our API implementations into first-order logic formulas and verifying these properties using Z3 [20].

5 Evaluation

Implementation. We prototype our ideas using the Barefoot Tofino switch [14], which can be programmed in P4-16. Tenants write their programs in P4-16 assuming full control over the switch resources allocated to them. Our linker reuses the P4-16 open-source compiler [44] to parse P4-16 source code for the tenant programs, check their resource usage, ensure the tenant ID is not modified, rename variables, and finally sandwich tenant programs between the system-level program (§4.2). Tenants modify their table entries by submitting requests to the control plane via a gRPC channel. The control plane uses Tofino [14] auto-generated Thrift APIs to carry out

	Program \ Resource Usage (% of total)	Exact Match Xbar	SRAM	Hash Bits Unit	Action Units	#Stages	Gateway tables	PHV
	System-level program (SYS)	0.26	0.31	0.6	0.26	8.33	0	14.58
Testcase 1	Firewall	1.82	1.25	1.92	1.3	33.33	3.125	7.44
	Source Routing	0.325	0.313	0.6	0.78	16.67	0.52	10.42
	SUM (SYS+Firewall+Source Routing)	2.4	1.875	3.125	2.343	58.33	**3.645**	**32.44**
	Merged program	1.95	1.25	1.923	1.823	33.33	**4.687**	**20.83**
Testcase 2	Load Balancing (LB)	0.52	0.31	1.24	1.04	25	1.04	7.44
	Link Monitor (LM)	0.65	0.52	0.78	0.26	16.7	1.56	18.15
	Multi-hop Route Inspection (MRI)	0.46	0.31	0.6	0.26	16.7	1.04	13.39
	SUM (SYS+LB+LM+MRI)	1.88	1.46	3.225	1.822	66.67	**3.64**	**53.57**
	Merged program	1.17	0.52	1.42	2.083	25	**5.73**	**33.33**
Testcase 3	NetCache	15.23	7.71	11.22	7.55	91.67	28.125	19.94
	NetChain	3.51	2.29	6.47	2.864	50	5.21	13.4
	SUM (SYS+NetCache+NetChain)	19.01	10.3125	18.29	10.677	150	**33.33**	**47.91**
	Merged program	18.36	9.69	17.09	9.896	100	**34.375**	**57.74**

Table 2: Resource usages of individual and merged programs on a Barefoot Tofino switch. The table size in each program is set to 1024 entries.

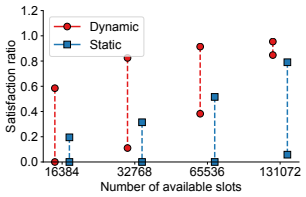

Figure 4: Satisfaction ratio of 64 heavy hitter detection tasks on Tofino.

tenant and system requests. The verification of our API (§4.3) takes 229s on an Intel Core i5-8300H @2.3GHz CPU. At run time, the resource allocator in the control plane periodically fetches utility statistics and validity status from the data plane via our verified APIs, which internally call the Thrift APIs.

Correctness of linker. To demonstrate that our linker provides isolation, we pick 5 tutorial P4 programs [46] together with NetCache [28] and NetChain [27] and create 3 groups of these programs. The system program provides basic forwarding and routing. We use Tofino's Packet Test Framework to inject packets and test the correctness of our data plane implementations. Our results (Table 2) show that all 3 groups of programs can be multiplexed on the same switch. Further, a single program only occupies a fraction of most hardware resources, showing how multitenancy can improve utilization. We also tested each of the programs within each group to check that it behaved as it would in a non-multiplexed setup.

For resource efficiency, the resource usage of the merged program should not be much more than the sum of resource usages of the system and tenant programs. Table 2 shows that this is largely the case, with the exception of gateway tables and PHVs. This is because (1) we add gateway tables to check which program is to be executed based on a packet's VID when merging programs and (2) Tofino's compiler needs to allocate additional tag-along PHV containers (i.e., containers that are passed unchanged from stage to stage) to support tenant programs that execute in downstream stages. Both are overhead: they aren't needed if each tenant program ran alone.

Run-time efficiency. To measure run-time efficiency, we use a metric called satisfaction ratio. Satisfaction ratio is a metric that measures the time fraction of a network task when its utility is above what it requires [42]. For our experiment, we assume that there are 64 tenants in total and every tenant launches their own heavy hitter detection task against a source IP address within its /6 subnet. These tasks arrive in a 10-minute period based on a Poisson process. Every task runs for 1 minute. For packet arrivals, we replay a 10-minute CAIDA trace [17] on a Barefoot Tofino switch. Figure 4 shows the satisfaction ratio of 64 heavy hitter detection tasks under different number of available slots in the one big array. The upper end of every vertical line is the mean while the lower end is the 5th percentile. The static allocation scheme will assign 1/64 of the total stateful memory to every task throughout the 10 minutes, while our approach takes realtime utility into consideration, leading to better satisfaction ratio.

6 Related Work

Hyper4 [26] and HyperV [57] virtualize software switches by declaring primitive tables and actions to emulate tenant programs. Compared with their methods, which incur high hardware resource overhead, the overhead of our approach is negligible (Table 2). P4Visor [58] enables lightweight virtualization through program analysis. However, its goal of sharing program logic between different modules does not readily apply in our context where each tenant runs its own program and is distrustful of other tenants. Further, Hyper4, HyperV, and P4Visor target software and FPGA-based switches, while our work targets pipeline-based ASICs. daPipe [13] shares our ideas of a common system program and compile-time merging, but targets incremental P4 programming rather than multitenancy. Further, it does not consider run-time dynamics like our run-time memory reallocation does.

7 Conclusion

Motivated by the emergence of high-speed programmable network devices, we propose mechanisms for multitenancy on these devices. These multitenancy mechanisms would allow such devices to be shared among different tenants of a cloud provider—in a manner similar to cloud compute and storage today. Our mechanisms use both compile-time and run-time approaches. Our preliminary results suggest that they provide both efficient resource utilization and inter-tenant isolation.

8 Discussion

Our goal with this paper is to start a discussion not just on mechanisms for multitenancy but on the vision of programmable network devices as a cloud service. What applications will benefit from network programmability in the public cloud? What will it take for cloud providers to provide this?

Interface questions. We have so far considered a model where cloud tenants submit entire programs to run on programmable devices. Is this the right interface? Should applications be written towards a higher-level API? This might enable simpler, less device-specific programming. It may also enable more lightweight isolation, e.g., paravirtualization.

Another traditional function of an operating system, beyond isolation and resource management, is to provide common services for applications (e.g., a file system). What is the equivalent for in-network applications? As one example, recent work proposes transparent swap-like access to remote memory as a way to build applications that exceed the on-chip memory capacity of a programmable switch [32]. Given the distributed nature of cloud systems, should the device OS provide abstractions for scalable and fault-tolerant storage?

Policy questions. What is an effective policy for partitioning multiple network resources between tenants? It should be both efficient and fair, and the diversity of resource types makes achieving this challenging. Naïve bin-packing is not strategy-proof: tenants may lie about their requirements to get higher allocations. Dominant-Resource Fairness (DRF) [24] offers one answer, but assumes Leontief utilities [3], which do not capture the typical diminishing returns in network applications [42]. Another potential policy could be a VCG auction [6]. But VCG needs precise mappings between resources and prices for each tenant, which are hard to obtain.

Hardware questions. Looking further into the future, how can changes to the hardware architecture of programmable network devices facilitate multitenancy? Our current design aims to work with today's programmable NICs and switches, which provide no such support, but future devices could allow more efficient and dynamic multitenancy. For example, adding a new tenant requires remerging and recompiling all programs. Support for partial reconfiguration, as provided in today's FPGAs [50], could allow us to restrict a new tenant's program to specific parts of the switch and reload only certain parts of the switch without impacting other tenants. As another example, dedicated hardware for address translation could reduce the overhead of our virtual stateful memory. What should device hardware designers take into account, and what are the costs of this additional hardware support?

Acknowledgments. We thank our shepherd Ymir Vigfusson and the reviewers for their feedback. This work is supported in part by NSF grants CRII-1755646, CNS-1813487, and CCF-1918757, and a Google Faculty Research Award.

References

[1] Amazon Virtual Private Cloud (VPC). https://aws.amazon.com/vpc/.

[2] In-band Network Telemetry. https://p4.org/assets/INT-current-spec.pdf.

[3] Leontief utilities - Wikipedia. https://en.wikipedia.org/wiki/Leontief_utilities.

[4] Naples DSC-100 Distributed Services Card. https://pensando.io/assets/documents/Naples_100_ProductBrief-10-2019.pdf.

[5] Terabit Ethernet - Wikipedia. https://en.wikipedia.org/wiki/Terabit_Ethernet.

[6] Vickrey–Clarke–Groves auction - Wikipedia. https://en.wikipedia.org/wiki/Vickrey%E2%80%93Clarke%E2%80%93Groves_auction.

[7] Virtual Network - Virtual Private Cloud | Microsoft Azure. https://azure.microsoft.com/en-us/services/virtual-network/.

[8] Virtual Private Cloud | Google Cloud. https://cloud.google.com/vpc.

[9] Vivado Design Suite User Guide | Partial Reconfiguration. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug909-vivado-partial-reconfiguration.pdf.

[10] Amazon EC2 F1 instances. https://aws.amazon.com/ec2/instance-types/f1/.

[11] Amazon Web Services. https://aws.amazon.com/.

[12] Microsoft Azure: Cloud Computing Services. https://azure.microsoft.com.

[13] BALDI, M. daPIPE a Data Plane Incremental Programming Environment. In *Proc. of the ACM/IEEE ANCS, 2019*.

[14] Barefoot Tofino. https://www.barefootnetworks.com/technology/.

[15] BENSON, T., AKELLA, A., AND MALTZ, D. A. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of the ACM IMC, 2010*.

[16] BOSSHART, P., GIBB, G., KIM, H.-S., VARGHESE, G., MCKEOWN, N., IZZARD, M., MUJICA, F., AND HOROWITZ, M. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *Proc. of the ACM SIGCOMM, 2013*.

[17] CAIDA datasets passive-2019. https://data.caida.org/datasets/passive-2019/.

[18] CHOLE, S., FINGERHUT, A., MA, S., SIVARAMAN, A., VARGAFTIK, S., BERGER, A., MENDELSON, G., ALIZADEH, M., CHUANG, S.-T., KESLASSY, I., AND ET AL. dRMT: Disaggregated Programmable Switching. In *Proc. of the ACM SIGCOMM, 2017*.

[19] DANG, H. T., BRESSANA, P., WANG, H., LEE, K. S., WEATHERSPOON, H., CANINI, M., PEDONE, F., AND SOULÉ, R. Network Hardware-Accelerated Consensus. Tech. Rep. USI-INF-TR-2016-03, Università della Svizzera italiana, May 2016.

[20] DE MOURA, L., AND BJØRNER, N. Z3: An Efficient SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008), Springer.

[21] Amazon EC2. https://aws.amazon.com/ec2.

[22] FIRESTONE, D., PUTNAM, A., MUNDKUR, S., CHIOU, D., DABAGH, A., ANDREWARTHA, M., ANGEPAT, H., BHANU, V., CAULFIELD, A., CHUNG, E., CHANDRAPPA, H. K., CHATURMOHTA, S., HUMPHREY, M., LAVIER, J., LAM, N., LIU, F., OVTCHAROV, K., PADHYE, J., POPURI, G., RAINDEL, S., SAPRE, T., SHAW, M., SILVA, G., SIVAKUMAR, M., SRIVASTAVA, N., VERMA, A., ZUHAIR, Q., BANSAL, D., BURGER, D., VAID, K., MALTZ, D. A., AND GREENBERG, A. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *Proc. of the USENIX NSDI, 2018*.

[23] Google Cloud Platform. https://cloud.google.com/.

[24] GHODSI, A., ZAHARIA, M., HINDMAN, B., KONWINSKI, A., SHENKER, S., AND STOICA, I. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proc. of the USENIX NSDI, 2011*.

[25] GREENBERG, A. The Art of Building a Reliable Cloud Network. Microsoft Research Faculty Summit, Aug. 2018. https://www.microsoft.com/en-us/research/video/the-art-of-building-a-reliable-cloud-network/.

[26] HANCOCK, D., AND VAN DER MERWE, J. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *Proc. of the ACM CoNEXT, 2016*.

[27] JIN, X., LI, X., ZHANG, H., FOSTER, N., LEE, J., SOULÉ, R., KIM, C., AND STOICA, I. NetChain: Scale-Free Sub-RTT Coordination. In *Proc. of the USENIX NSDI, 2018*.

[28] JIN, X., LI, X., ZHANG, H., SOULÉ, R., LEE, J., FOSTER, N., KIM, C., AND STOICA, I. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proc. of the ACM SOSP, 2017*.

[29] JOSE, L., YU, M., AND REXFORD, J. Online Measurement of Large Traffic Aggregates on Commodity Switches. In *Proc. of the USENIX Hot-ICE, 2011*.

[30] KHAWAJA, A., LANDGRAF, J., PRAKASH, R., WEI, M., SCHKUFZA, E., AND ROSSBACH, C. J. Sharing, Protection, and Compatibility for Reconfigurable Fabric with Amorphos. In *Proc. of the USENIX OSDI, 2018*.

[31] KIM, D., MEMARIPOUR, A., BADAM, A., ZHU, Y., LIU, H. H., PADHYE, J., RAINDEL, S., SWANSON, S., SEKAR, V., AND SESHAN, S. Hyperloop: Group-based NIC-offloading to Accelerate Replicated Transactions in Multi-tenant Storage Systems. In *Proc. of the ACM SIGCOMM, 2018*.

[32] KIM, D., ZHU, Y., KIM, C., LEE, J., AND SESHAN, S. Generic External Memory for Switch Data Planes. In *Proc. of ACM HotNets, 2018*.

[33] LERNER, A., HUSSEIN, R., AND CUDRÉ-MAUROUX, P. The Case for Network Accelerated Query Processing. In *Proc. of the ACM CIDR, 2019*.

[34] LI, J., MICHAEL, E., AND PORTS, D. R. K. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proc. of the ACM SOSP, 2017*.

[35] LI, J., MICHAEL, E., SHARMA, N. K., SZEKERES, A., AND PORTS, D. R. K. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proc. of the USENIX OSDI, 2016*.

[36] LI, X., SETHI, R., KAMINSKY, M., ANDERSEN, D. G., AND FREEDMAN, M. J. Be Fast, Cheap and in Control with SwitchKV. In *Proc. of the USENIX NSDI, 2016*.

[37] LiquidIO Smart NICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics.html.

[38] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading Distributed Applications onto SmartNICs Using IPipe. In *Proc. of the ACM SIGCOMM, 2019*.

[39] LIU, Z., BAI, Z., LIU, Z., LI, X., KIM, C., BRAVERMAN, V., JIN, X., AND STOICA, I. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proc. of the USENIX FAST, 2019*.

[40] Mellanox High-performance Programmable SmartNICs. https://www.mellanox.com/products/smartnic.

[41] MIAO, R., ZENG, H., KIM, C., LEE, J., AND YU, M. Silkroad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proc. of the ACM SIGCOMM, 2017*.

[42] MOSHREF, M., YU, M., GOVINDAN, R., AND VAHDAT, A. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of the ACM SIGCOMM, 2014*.

[43] P4 Language. https://p4.org/.

[44] P4-16 Reference Compiler. https://github.com/p4lang/p4c.

[45] P4 Runtime. https://p4.org/p4-runtime/.

[46] P4 Tutorial. https://github.com/p4lang/tutorials.

[47] Pensando Announces P4-programmable Platform and Joins P4 Community. https://p4.org/p4/pensando-joins-p4.html.

[48] Amazon S3. https://aws.amazon.com/s3/.

[49] SAPIO, A., CANINI, M., HO, C.-Y., NELSON, J., KALNIS, P., KIM, C., KRISHNAMURTHY, A., MOSHREF, M., PORTS, D. R. K., AND RICHTARIK, P. Scaling Distributed Machine Learning with In-Network Aggregation. Tech. rep., KAUST, Feb. 2019.

[50] SAQUETTI, M., BUENO, G., CORDEIRO, W., AND AZAMBUJA, J. R. Hard Virtualization of P4-Based Switches with VirtP4. In *Proc. of the ACM SIGCOMM Conference Posters and Demos, 2019*.

[51] SHARMA, N. K., KAUFMANN, A., ANDERSON, T., KIM, C., KRISHNAMURTHY, A., NELSON, J., AND PETER, S. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *Proc. of the USENIX NSDI, 2017*.

[52] SHARMA, N. K., LIU, M., ATREYA, K., AND KRISHNAMURTHY, A. Approximating Fair Queueing on Reconfigurable Switches. In *Proc. of the USENIX NSDI, 2018*.

[53] SIVARAMAN, A., CHEUNG, A., BUDIU, M., KIM, C., ALIZADEH, M., BALAKRISHNAN, H., VARGHESE, G., MCKEOWN, N., AND LICKING, S. Packet Transactions: High-Level Programming for Line-Rate Switches. In *Proc. of the ACM SIGCOMM, 2016*.

[54] SIVARAMAN, A., SUBRAMANIAN, S., ALIZADEH, M., CHOLE, S., CHUANG, S.-T., AGRAWAL, A., BALAKRISHNAN, H., EDSALL, T., KATTI, S., AND MCKEOWN, N. Programmable Packet Scheduling at Line Rate. In *Proc. of the ACM SIGCOMM, 2016*.

[55] SONG, H. Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane. In *Proc. of the ACM HotSDN, 2013*.

[56] Stingray SmartNIC Adapters and IC. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[57] ZHANG, C., BI, J., ZHOU, Y., DOGAR, A. B., AND WU, J. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *Proc. of the IEEE ICCCN, 2017*.

[58] ZHENG, P., BENSON, T., AND HU, C. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *Proc. of the ACM CoNEXT, 2018*.

[59] ZHU, H., BAI, Z., LI, J., MICHAEL, E., PORTS, D. R. K., STOICA, I., AND JIN, X. Harmonia: Near-linear Scalability for Replicated Storage with In-network Conflict Detection. *Proc. of VLDB Endowment, 2019*.