LEVERAGING APPLICATION KNOWLEDGE FOR HIGH-PERFORMANCE NETWORKS: FROM PRESCHEDULED CIRCUIT SWITCHING TO RUNTIME VERIFICATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sundararajan Renganathan
February 2026

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Nick McKeown) Principal Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Balaji Prabhakar)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

(Christos Kozyrakis)

Approved for the Stanford University Committee on Graduate Studies

Abstract

Modern high-performance networks face two fundamental challenges: ensuring correct operation at scale and efficiently supporting specialized workloads. This thesis addresses both challenges by demonstrating how leveraging application-specific knowledge can yield significant improvements in network design and operation. We present three systems that use different forms of application knowledge to optimize network behavior, ranging from runtime verification to circuit-switched architectures for AI training.

The first contribution, Hydra, addresses the challenge of verifying network correctness in real time. We present a domain-specific language and compiler that enables operators to specify network-wide properties and verify that every packet is correctly processed against these specifications at line rate. The compiler generates executable P4 code that runs alongside forwarding logic on programmable switches, enabling scalable runtime verification without centralized bottlenecks. The second contribution, Chronos, explores circuit-switched fabrics for large language model training by exploiting the predictability of training traffic patterns. We develop techniques to analyze training code and generate switch schedules for the entire training run, replacing packet-switched fabrics with pre-scheduled crossbar switches to reduce power consumption and increase switching capacity. For unpredictable traffic patterns such as those in Mixture-of-Experts models, we employ Birkhoff-von Neumann decomposition to schedule circuits on demand. The third contribution, BhaVaN, provides the hardware foundation needed to support dynamic traffic in circuit-switched networks. We present a hardware algorithm that performs Birkhoff-von Neumann decomposition in nanoseconds using wrapped wavefront arbiters with bitwise parallelism, reducing runtime complexity from $O(N^{4.5})$ to $2N$ clock cycles for an $N \times N$ matrix. Collectively, these systems demonstrate that leveraging application knowledge—whether about network policies, training patterns, or traffic matrices—enables practical solutions to hard problems in high-performance networking.

Acknowledgments

A PhD is often described as a long and at times seemingly never-ending journey; mine has certainly felt that way. As I sit down to write this thesis, I am deeply grateful for the mentors, collaborators, friends, and family who have walked this path with me. This document may carry my name alone, but it is, in many ways, the product of a whole community. Their influence goes far beyond what I can express on this page, but I will nonetheless try to thank them here.

First and foremost, my deepest gratitude is to my advisor, Nick McKeown, who taught me that it takes bold bets to make a dent in systems and networking research, that conventional wisdom is always fair game to question even if it means swimming against the tide, and that aspiring to change the world is exactly what a PhD student should do. Beyond his intellectual guidance, he consistently went a step further to mentor me through the peaks and troughs of this journey—especially in moments when results were unclear, papers were rejected, or I doubted my ability to make progress. His patience, candor, and unwavering belief that the work (and I) could get somewhere meaningful have left a lasting mark on how I think about research and about myself. I will be forever grateful for that.

I am also indebted to Balaji Prabhakar and Christos Kozyrakis for serving on my defense committee and for carefully reading this thesis. Their early feedback on my work was crucial in shaping it into its final form and pushed me to sharpen both the ideas and their presentation. In addition, I am grateful to Sachin Katti and Ewart Thomas for generously agreeing to serve on my defense committee on short notice.

I have been fortunate to share this journey with wonderful labmates—Serhat Arslan, Evgenya Pergament, Bruce Spang, Stephen Ibanez, Alex Mallery, Theo Jepsen, and Catalin Voss—who welcomed every half-baked idea, debugged both code and thinking with me, and made the pandemic years far more bearable than they might have been. The conversations in the lab, the shared frustrations, and the small victories we celebrated together are among my most cherished memories of Stanford.

I have also had the privilege of working with incredible collaborators. Foremost among them, I would like to deeply thank Nate Foster for serving as my pseudo-advisor (or should I say foster-advisor?!) when Nick and Sachin were on leave from Stanford and for the time I spent working

with his group at Cornell. In a period that could easily have felt unmoored, he gave my PhD a sense of continuity and direction. The Hydra project would not have been possible without his astute stewardship and patience as we worked to land it. That project deeply shaped the subsequent directions I pursued in my PhD, giving me the confidence to tackle hard problems rather than limit myself to low-hanging fruit. I am also grateful to Benny Rubin, Carmelo Cascone, Charles Chan, Pier Ventre, and Daniele Moro, who recognized Hydra's potential early and were happy to jump in for late-night debugging sessions as we made progress. I also thank KK Yap and Nandita Dukkipati for a wonderful internship experience at Google, where I made connections I continue to lean on to this day.

Working on the DARPA Pronto demos that showcased Hydra's runtime network verification approach on a swarm of drones (flying over Lake Lagunita during one of its not-so-dry phases!) by preventing DoS and exfiltration attacks was one of the most rewarding experiences of my PhD. For making that possible, I would like to thank Nick, Nate, Jennifer Rexford, Hyojoon Kim, Guru Parulkar, Oguz Sunay, and Bruce. Those frantic days of integration and live demos taught me as much about collaboration and trust as they did about systems.

My research journey began well before I started my PhD at Stanford, when I spent two wonderful years at Microsoft Research India. I thank Venkat Padmanabhan for giving me my first real deep dive into research; his energy and attention to detail rubbed off on me and set the bar for what serious, careful work should look like. My time at MSR India was instrumental in my decision to pursue a PhD. Someone back then said that the lows of research can be really low, but the delirious highs of making a breakthrough more than make up for them, and I still consider this true. I also thank my other mentors from MSR—Ganesh Ananthanarayanan, Sreangsu Acharyya, Muthian Sivathanu, Ramachandran Ramjee, and Ranjita Bhagwan—for their guidance, patience with my early missteps, and for showing me different ways to think about systems problems. My time there would not have been the same without friends—Sachin Ashok, Venkatesh Balasubramanian, Apurv Mehra, Sudheesh Singanamalla, Aditya Kusupati, and Viral Mehta—who made the atmosphere collegial and shared in the frustrations and joys that first-time research entails.

Outside of work, I was lucky to find a circle of friends at Stanford—Shreya Singh, Nishant Rai, Kopal Nihar, Neil Perry, Sweta Soni, Ashwin Kanhere, Advay Pal, Kumar Ayush, Abhishek Sinha, Chirag Pabbaraju, Tara Mina, Aditi Partap, Sanjari Srivastava, William Meng, Prabhat Agarwal, Yash Bhartia, Haroun Habeeb, Soham Gadgil, Tanushree Lahiry, Sanket Gupte, Aditya Gera, Rishabh Seth, Shantanu, and Sanchita. They turned what could have been an isolating few years into something that felt like a community. They made the pandemic much more bearable, provided support when the going was tough, and shared in my laughter during the good times. Many of my happiest memories from this period have nothing to do with research papers, and everything to do with evenings and weekends spent with them.

During my PhD I also stayed in close touch with friends from my undergrad at IIT Kanpur. Rachit

Agarwal, Siddhant Dangi, Siddharth Tanwar, Pranav Vaish, Anand Singh Kunwar, Rohan Bavishi, and Abhimanyu Goyal were always there to take my mind off research, to put things in perspective, and to remind me that there was life before and after any particular deadline or result. I am grateful that we stayed connected over the years despite the different paths we took after undergrad. The same can be said of my friends from high school—Divyanshu Verma, Mihir Bharambe, and Aditya Kanthale—whom I could count on for much-needed trips down memory lane to the more carefree and pristine times of high school.

A special thank you is reserved for Shashank Bhushan and Shishir Patil, with whom my closeness has led to comments that we are "brothers from another mother." They have been constant companions through the lows of my PhD journey and celebrated every small win like it was their own. They listened to countless rants, sat with me through uncertainty, and refused to let me give up when progress felt painfully slow. Without them, I am fairly sure this thesis would not exist in its current form. I can only hope to one day repay, even in part, the support they showed me when there seemed to be no way out.

Finally, I would like to thank my family—my sister, Jayashree, and my parents, Rohini (*amma*) and Renganathan (*appa*). It often felt during my PhD that they were on this journey with me despite being 10,000 miles away in India. They carried my worries across time zones, listened patiently to my ups and downs, and never stopped believing that I would find a way through. They taught me the importance of hard work, the belief that things will get better no matter how bad they seem, and made countless sacrifices along the way so that I could pursue opportunities they themselves never had. Any strength I had in finishing this thesis comes from them. If there is any pride in these pages, it belongs to them first. To them, I dedicate this thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern high-performance networks operate at unprecedented scale and complexity. Data center networks connect millions of servers, cellular networks coordinate thousands of base stations, and AI training clusters orchestrate hundreds of thousands of accelerators. Despite significant advances in networking technologies, two challenges persist: ensuring that networks behave correctly at runtime, and efficiently supporting specialized workloads with predictable characteristics. This thesis demonstrates that leveraging application-specific knowledge—whether about desired network properties, or communication patterns—enables practical solutions to both challenges.

1.1 The Challenge of Network Verification

Operating networks correctly remains difficult, particularly at scale. Misconfigurations, software bugs, hardware faults, and security breaches occur frequently in production networks [79, 132, 5]. Networks must forward packets according to complex specifications while maintaining tenant isolation, enforcing security policies, and ensuring quality of service guarantees. As networks grow larger and more programmable, not only does the likelihood of failures increase, but the blast radius of failures also increases. Not only that, but the attack surface is also proportionally larger.

1.1.1 Limitations of Static Verification

Prior work has developed sophisticated static verification techniques. Static checkers take snapshots of network forwarding state—device configurations or forwarding rules—and build mathematical models of network behavior. These models can verify reachability [76], check routing properties [33, 15], verify network-wide invariants [59, 8], and validate cloud contracts [19]. Tools like Veriflow [61], NetPlumber [58], and those based on atomic predicates [136] have seen deployment in production networks and prevented numerous outages.

However, static verification is quite limited. First, it is hard to collect complete complete data plane snapshots at scale [54, 141]. Second, static checkers are restricted to analyzing stable configurations and cannot reason about transient states during updates [15]. Most importantly, static checkers verify models rather than actual network behavior. If the model does not capture some aspect of reality—a bug in switch software, an error in NIC drivers, or a hardware fault—the checker may miss it completely. Experience by large operators suggests that as static verification has matured, failures increasingly stem from implementation bugs rather than configuration errors [5, 79, 132].

1.1.2 Runtime Verification Approaches

Runtime verification systems address these limitations by checking actual network behavior. Active testing approaches send probe packets and verify them against specifications [140, 83, 23, 112, 106]. However, probes only detect bugs along paths they exercise, and generating comprehensive test coverage for both the topology and switch code paths remains challenging.

Passive monitoring approaches collect telemetry from production traffic. Systems like NetSight [47], VeriDP [143], and PathDump [118] attach metadata to packets and analyze it offline at centralized servers. Confluo [60] and SwitchPointer [119] provide high-throughput telemetry collection. While these systems provide valuable visibility, centralized analysis becomes a bottleneck at scale. Processing telemetry from thousands of switches handling terabits per second of traffic overwhelms centralized servers and introduces latency between when violations occur and when they are detected.

This thesis presents a different approach: distributed, in-network verification that checks every packet at line rate against formal specifications. By distributing verification across switches and performing checks in the data plane, we achieve both scalability and real-time enforcement.

1.2 The Opportunity in Specialized Networks

While general-purpose data centers must handle unpredictable, heterogeneous workloads, specialized networks exhibit characteristics that enable fundamentally different architectures. AI training clusters provide a compelling example of how exploiting workload knowledge can inform network design.

1.2.1 Predictability in AI Training

Large language model training employs various forms of parallelism: data parallelism [100], tensor parallelism [111], pipeline parallelism [49, 82], sequence parallelism [64], and context parallelism [71]. Each parallelism strategy requires communication between specific accelerator groups in predetermined patterns. Developers implement these patterns using collectives [86]—structured operations like all-reduce, all-gather, and reduce-scatter—rather than arbitrary point-to-point communication.

Recent work has characterized training communication patterns and found them to be largely

predictable [97, 69]. The model developer determines when messages are sent, their sizes, and the participating accelerators. This predictability contrasts sharply with general data center workloads where traffic patterns emerge from thousands of independent applications.

Current training clusters employ packet-switched networks: scale-up networks connect accelerators within racks (NVLink [89], TPU ICI [56], AMD Infinity Fabric [113], AWS NeuronLink [109]), while scale-out networks connect racks (InfiniBand [88], RoCE [41], UEC [123]). These packet-switched fabrics must be over-provisioned to avoid congestion [41], yet still experience hotspots that delay training [96]. Variable latency is particularly problematic because synchronous collective operations wait for the slowest transfer to complete.

1.2.2 Circuit Switching for Known Traffic

Packet switching excels at handling unpredictable traffic from heterogeneous applications. However, training workloads exhibit neither unpredictability nor heterogeneity. This raises a fundamental question: are packet switches optimal for carrying predictable, structured traffic?

Circuit switches offer an alternative. A circuit switch is simply a crossbar where each input connects to at most one output during each time slot, with connections specified by permutation matrices. Circuit switches can be dramatically simpler than packet switches. Analysis of switch architecture [20] shows that packet switches dedicate approximately 30% of die area to serial I/O, 50% to packet processing pipelines, and 20% to buffers and traffic management. Circuit switches eliminate packet processing logic, lookup tables, and buffers, freeing substantial die area for either power reduction or increased I/O capacity. Additionally, circuit switches provide deterministic, minimal latency without the variable queuing delays inherent in packet switching.

Optical circuit switches bring additional benefits. Google's deployment of optical circuit switches for cluster provisioning [72] demonstrated the viability of dynamically reconfigurable optical interconnects at scale. Prior work has explored optical circuit switching for datacenter networks [94, 92, 125, 13], though not specifically for training workloads.

The challenge is handling unpredictable traffic. While most parallelism strategies have known patterns, expert parallelism in Mixture-of-Experts (MoE) models [32, 78] generates input-dependent traffic. Routing decisions depend on model activations and cannot be predetermined. Additionally, control traffic, checkpoint operations, and failure recovery create unpredictable demands. Supporting these workloads requires scheduling circuit switches in real time, which demands solving combinatorial optimization problems on microsecond timescales.

1.3 Contributions

This thesis makes three primary contributions demonstrating how application knowledge enables practical solutions to networking challenges.

1.3.1 Hydra: Effective Runtime Network Verification

We present Hydra, a system that verifies every packet traversing a network against formal specifications at line rate. Hydra introduces Indus, a domain-specific language for expressing network-wide properties. Unlike P4 or other switch-level languages, Indus operates at the network level with three key abstractions: an `init` block that executes when packets enter the network, a `telemetry` block that collects information at each hop, and a `checker` block that evaluates predicates at the last hop.

We develop a compiler that translates Indus programs into P4 code that runs alongside forwarding logic on programmable switches. The compiler generates switch-specific checking code that executes independently of forwarding, providing fault isolation—a bug affecting both forwarding and checking is less likely than a bug affecting only one. By performing checks in the data plane rather than at centralized servers, Hydra achieves scalability without bottlenecks.

We prove that Indus can express all properties encodable in Linear Temporal Logic over finite traces (LTL_f), providing theoretical grounding for its expressiveness. We implement and deploy Hydra checkers in Aether [35], an open-source cellular platform, where we detect a subtle bug in the 5G mobile core involving packets egressing through incorrect ports—behavior that violated network invariants but would be difficult to detect with static analysis alone.

Performance evaluation on Intel Tofino switches [52] demonstrates that Hydra adds minimal overhead. For properties including path validation and slice isolation, checking adds less than a microsecond of latency and often requires no additional pipeline stages. These results establish that comprehensive runtime verification is practical for modern programmable switches.

1.3.2 Chronos: Pre-Scheduled Circuit Switching for LLM Training

We present Chronos, a system that replaces packet-switched fabrics in training clusters with pre-scheduled circuit switches. The key insight is that most training communication can be scheduled before training begins by analyzing the training code. We develop techniques to extract collective operations from training programs and generate complete schedules of permutation matrices that configure crossbar switches throughout the training run.

For hierarchical network topologies with multiple switch tiers, we decompose network-wide permutations into local per-switch permutations. When fabric tiers have equal capacity (non-oversubscribed networks), decomposition is straightforward—traffic routes through top tiers and back. For oversubscribed networks or power optimization, we can exploit locality and schedule shorter time slots locally than globally.

The architecture employs time-synchronized crossbar switches with no packet buffers, forwarding tables, or packet processing logic. This simplification enables lower power consumption, increased capacity through additional I/O, and deterministic low latency. The centrally coordinated timing also simplifies failure detection—stragglers and failed components are more easily distinguished in synchronous systems.

For unpredictable traffic including MoE expert parallelism, control messages, and failure recovery, Chronos employs on-demand scheduling using Birkhoff-von Neumann decomposition. This algorithm converts traffic demand matrices into permutation sequences, but traditional implementations are too slow for real-time use. Chronos addresses this with the BhaVaN algorithm described next.

1.3.3 BhaVaN: Fast Birkhoff-von Neumann Decomposition in Hardware

We present BhaVaN, a hardware algorithm that performs Birkhoff-von Neumann decomposition in nanoseconds rather than milliseconds. Given a doubly stochastic $N \times N$ matrix, Birkhoff-von Neumann decomposition produces a sequence of permutation matrices with weights such that the original matrix is their weighted sum. The algorithm proceeds iteratively: perform bipartite matching, find the minimum value along matched edges, subtract that value, and record the permutation. Theorems by Birkhoff [18] and Hall [46] guarantee validity for doubly stochastic matrices when using maximum cardinality matching.

Traditional implementations using Hopcroft-Karp matching [48] with $O(N^{2.5})$ complexity achieve overall $O(N^{4.5})$ runtime. Software implementations require hundreds of milliseconds for 128×128 matrices typical of MoE deployments. This latency is fundamentally unsuitable for real-time circuit switching, which requires microsecond-scale scheduling.

BhaVaN achieves dramatic speedup through parallelism and specialized hardware. The algorithm employs Wrapped Wavefront Arbiters (WWFA) [117] for bipartite matching, providing $O(N)$ matching time rather than $O(N^{2.5})$. While WWFA produces maximal rather than maximum matchings, potentially requiring more iterations, the vastly faster per-iteration time yields substantial overall improvement.

The critical innovation is bitwise parallelism. For matrices with M-bit integer values, traditional implementations process bits sequentially. BhaVaN instantiates M parallel WWFA units, one per bit position, processing all bits simultaneously. For an $N \times N$ matrix with M-bit values, BhaVaN completes in $2N - 1$ clock cycles independent of M. With pipelining, new decompositions complete every N cycles.

We implement BhaVaN in Verilog and synthesize using TSMC 16nm libraries. For 128×128 matrices, decomposition completes in 256 nanoseconds at 1 GHz clock frequency. The hardware footprint is modest—less than 1 mm^2 for 128×128 matrices with 16-bit values. Performance and area scale favorably to newer process nodes.

We compare BhaVaN with alternative matching algorithms including PIM [9], iSLIP [77], and maximum matching approaches. While maximal matching may produce slightly longer permutation sequences than maximum matching, cycle times (sum of weights) remain comparable. The orders-of-magnitude reduction in computation time far outweighs minor increases in sequence length.

BhaVaN enables circuit switching for arbitrary dynamic traffic. The algorithm is general enough to handle any unpredictable traffic pattern in training clusters—MoE communication, control messages,

checkpoint traffic, and error recovery—making circuit switching viable for the full spectrum of datacenter workloads.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 presents Hydra, including the Indus language design, compiler architecture, deployment experiences, and performance evaluation on programmable switches. Chapter 3 presents Chronos, including techniques for extracting communication patterns from training code, pre-scheduled circuit-switched fabric design, and mechanisms for handling failures. Chapter 4 presents BhaVaN, including the algorithm design employing bitwise parallelism and WWFA, hardware implementation in Verilog, and comparisons with alternative matching algorithms. Chapter 5 concludes with reflections on broader implications and directions for future work.

Chapter 2

Runtime Verification with Programmable Switches

2.1 Introduction

As discussed in Chapter 1, ensuring network correctness remains a fundamental challenge despite advances in networking technologies. At first glance, most packet-switched networks appear simple: each device implements straightforward tasks like looking up headers in routing tables, filtering packets using access control lists, and adding or removing tunneling headers. However, operating a network correctly at scale is difficult. Faults, outages, performance degradation, and security breaches occur frequently in practice, for reasons ranging from simple misconfigurations to pernicious hardware and software bugs. Misconfigurations and bugs can appear anywhere—the control plane or the data plane, fixed-function switches or programmable switches, conventional NICs or smart-NICs, the end host networking stack, and so on.

Prior work has proposed methods and tools to check if a network correctly forwards traffic according to a formal specification. For example, static checkers take snapshots of the network forwarding state (e.g., device configurations or forwarding rules) to build mathematical models of network behavior. These models can be used to verify cloud contracts [19], to answer "what if" questions about router configurations [33, 15], and to verify network-wide properties like connectivity, waypointing, and freedom from loops [76, 59, 8, 61, 58, 136]. Despite enjoying great success, they have well-known limitations regarding scalability [54], the complexity of collecting data plane snapshots [141], and the restriction to stable configurations [15]. Moreover, there is a growing sense (e.g., at Google [5], Facebook [79], and Microsoft [132]) that the success of static checking has shifted the goalposts—the most important failures now often relate to switch hardware and software bugs rather that simple misconfigurations.

Perhaps the most important limitation of static checkers, however, is that they rely on an accurate model of the network. Hence, static checkers can make mistakes if the abstract models they rely upon do not reflect the "ground truth" experienced by packets traveling through the data plane. For example, a static checker might deduce that an end-to-end path exists (based on its model of the forwarding state), but due to bugs in the end host networking stack or some other part of the network that the static checker does not model (e.g., the low-level driver code for the switches) the packet might actually follow a different path. In this situation, ironically, rudimentary tools like `ping` or `traceroute` can successfully detect a bug that the static checker cannot! This modelling limitation exists irrespective of how the static checker builds its model—if any aspect of the network's behavior is not reflected in the model, then some bugs may go undetected.

In contrast, runtime verification systems can verify the behavior of the network in real time, directly in the data plane. One approach is to send special probe packets and check them against a model [140, 83, 23, 112, 106]. However, this technique only works if the probe packets test all the paths (in the topology and in the code). A second approach is to attach additional information or telemetry data to real data packets, which are collected and analyzed offline at a centralized server [143, 47, 60, 119, 118]. This technique is hard to scale for large or fast networks, because the centralized server quickly becomes the bottleneck.

This chapter addresses the following question:

> *Can a network check that every packet is correctly processed, in real-time, against a specification?*

Our approach is *Hydra*, a system that uses ideas from the field of runtime verification [14] and applies them to networking. Rather than analyzing idealized models or performing post-hoc analysis of telemetry, Hydra allows an operator to verify that each packet traversing the network is processed according to a formal specification. Properties are specified in *Indus*, a domain-specific language (DSL) we designed.

Indus is designed to require little to no understanding of the forwarding specification, and operates at a higher level of abstraction. In fact, it reads like typical imperative programming languages that operators are already familiar with. A key distinguishing feature of Indus is that it models network-wide, stateful properties using *telemetry* (comprising packet state, switch-local state, and control-plane state) and *checkers*, which are predicates over telemetry that determine whether a packet should be forwarded, rejected, or reported to the control plane. Indus operates at a higher layer of abstraction than existing DSLs (e.g., P4, eBPF, and DPDK), enabling operators to focus on higher-level behaviors, without concern for how and where they are implemented, or what devices they are compiled to.

Hydra verifies every packet by collecting telemetry data, adding it to packets as they make their way through the network. Indus only requires programmers to specify what telemetry should be collected at each hop and what the predicate on that telemetry should be. By checking each

```
/* Variable declarations */
control dict<bit<8>,bit<8>> tenants;
tele bit<8> tenant;
header bit<8> in_port;
header bit<8> eg_port;

/* Code blocks */
init { /* Executes at first hop */
  tenant = tenants[in_port];
}
telemetry { /* Executes at every hop */ }
checker { /* Executes at the last hop */
  if (tenant != tenants[eg_port]) { reject; }
}
```

Figure 2.1: Indus program for bare-metal multi-tenancy.

packet, on switch, without the need for a central server, Hydra is inherently scalable and can enforce properties in real time.

The key contributions of this chapter are as follows. First, Hydra, a practical system for checking network-wide properties in real time at line rate (Section 2.2). Second, Indus, a domain-specific language that allows operators to specify runtime verification policies concisely (Section 2.3). Third, the Indus Compiler that generates switch-specific checking code that executes independently of the forwarding code (Section 2.4). Fourth, we demonstrate that Hydra can find bugs in real-world networks by building a working prototype and using it to implement path validation for source routing and to detect a subtle bug in Aether [35] (Section 2.5). Fifth, we assess the expressiveness of Indus from both theoretical and practical perspectives, showing that Indus can express all properties encodable in Linear Temporal Logic over finite traces (LTL_f), and we develop Indus programs for a range of properties studied previously in the network verification literature (Section 2.6). Finally, we evaluate the overheads of Hydra on Tofino switches [52], finding that the costs of implementing Indus checkers are modest, whether measured in terms of pipeline resources, packet-processing latency, or throughput (Section 2.6).

2.2 Hydra By Example

In this section, we introduce a series of examples based on real-world scenarios where there is a need for verification. These examples showcase how Hydra takes runtime verification (RV) ideas and applies them to networking, a hitherto underexplored avenue for said ideas. Each example first describes the real-world scenario, then gives an intuitive description of the property being verified, and then presents a program that expresses the property in Indus.

Bare-metal multi-tenancy. In bare-metal cloud services, tenants have full control over physical servers, including the NIC and host networking stack. To ensure tenant isolation, the Top-of-Rack (ToR) switch is typically programmed with functions such as Virtual Routing and Forwarding (VRF)

tables and VXLAN encapsulation [11]. In this setup, all traffic sent and received through a given port connected to a physical server is expected to belong to the same tenant. If any packet crosses between supposedly isolated tenants, the cloud provider risks losing business and trust. The Indus program in Figure 2.1 enforces network-wide, per-port traffic isolation.

There are two important things to note about this program. First, while Indus is a specification language, programs read more like a scripting language than a formula in logic. This is deliberate: The choice to design a new DSL, rather than re-using an existing logical framework (e.g., Linear Temporal Logic), to avoid the well-known challenges that arise when programmers are asked to write specifications. Second, unlike existing networking DSLs like P4, which captures the functionality of a single switch, Indus models the end-to-end behavior of the network. Hence, it can be used to express network-wide properties—*e.g.*, that each packet must enter and exit via ports on the same tenant.

More formally, an Indus program comprises three blocks. The `init` block executes when a packet enters the network, at the first hop, before it has undergone any other processing. The `telemetry` block executes at every hop, including the first and last hops.[1] The `checker` block executes only at the *last* hop, before the packet exits the network (*e.g.*, in the egress pipeline of the last switch). It executes a predicate on the collected telemetry, which can either come from the `init` or `telemetry` block, to determine whether the packet should be halted ("`reject`"), allowed to proceed, or allowed to proceed but with a report generated ("`report`").

Indus supports several different kinds of variables, each related to how they are used: `tele` variables are carried in the packet, while `control` variables are switch-local state that are managed by the control plane. In this example, the `tenants` control variable is realized as a table that associates switch ports to tenants. The `tenant` telemetry variable records the tenant associated with the original ingress port in the packet. At the last hop, the `checker` block verifies that the ingress and egress ports were associated with the same tenant, and rejects the packet if not.

Load Balancing. For the next example, consider a tiered data center network with servers connected to ToR switches. Data center operators typically spread traffic across multiple paths (e.g., at the granularity of flows [4], flowlets [7] or even individual packets) to balance the load, which reduces congestion. In our example, the program will check that the actual usage of the uplink switch ports is approximately balanced, within a given threshold.

Figure 2.2 shows how load-balancing can be specified in Indus in an intuitive manner. To keep the example simple, the focus is on load balancing across just two ports (`left_port` and `right_port]`), but the program generalizes to any number of ports in a straightforward manner. Note that load balancing is verified on a *per-packet* basis, even if the implementation of load balancing is performed at per-flow granularity. This approach is more scalable than polling each switch for per-port utilization information and then checking whether the load is imbalanced. To implement the desired functionality, the Indus program uses `sensor` variables, which aggregate telemetry data across multiple packets

[1] In this example the telemetry block is empty but it will be used in other examples.

```
sensor bit<32> left_load = 0;
sensor bit<32> right_load = 0;
control left_port;
control right_port;
control thresh;
control dict<bit<8>,bool> is_uplink;
tele bit<32>[15] left_loads;
tele bit<32>[15] right_loads;
header bit<8> eg_port;

init { }
telemetry {
  if (is_uplink[eg_port]) {
    if (eg_port == left_port) {
      left_load += packet_length;
    }
    elsif (eg_port == right_port) {
      right_load += packet_length;
    }
  }
  left_loads.push(left_load);
  right_loads.push(right_load);
}
checker {
  for (left_load, right_load in left_loads, right_loads) {
    if (abs(left_load - right_load) > thresh) {
      report;
    }
  }
}
```

Figure 2.2: Indus program for data center load balancing.

using switch-local state, and a non-trivial `telemetry` block, which records the total amount of data transmitted on each port in `tele` variables. The `checker` block iterates over the telemetry and flags a `report` if it detects an imbalance above a fixed threshold. It is worth noting that the left and right port numbers, as well as the load imbalance threshold are `control` variables. Hence, these values can be changed on the fly, without having to recompile the Indus program.

As shown in this example, telemetry is collected as the packet makes its way through the network in the form of a list, and only the check is performed at the last hop. This provides a nice abstraction, similar to that of classical runtime verification, where the Indus program only needs to specify a read-only trace that is collected as the packet flows through the network (telemetry block) and a predicate on that trace (checker block). Enforcing the check at the last hop has the nice property of moving programmability from the core to the edge of the network, where the functionality could be implemented on a smartNIC or even in the kernel. The design decision is elaborated on in Section 2.4.

Stateful Firewall. Figure 2.3 is a program to enforce the property that packet flows can only enter the network if a device inside the network initiated the communication. To accomplish this, the control plane installs rules in the reverse direction when it sees a packet in the forward direction. As described earlier, Indus programs are coupled to the network topology, which might mandate that packets enter and leave through a designated choke point. However, this Indus program is generic enough to check this property in other topologies. For example, every edge switch could be a firewall,

```
control dict<(<bit<32>,bit<32>),bool> allowed;
tele bool violated = false;

header bit<32> ipv4_src;
header bit<32> ipv4_dst;

init { /* Checks if packet is allowed to enter */
  if (!allowed[(ipv4_src,ipv4_dst)]) {
    violated = true;
  }
}
telemetry { /* Checks if packet on reverse direction has been seen */
  if (last_hop && !allowed[(ipv4_dst, ipv4_src)]) {
    report((ipv4_dst,ipv4_src));
  }
}
checker {
  if (violated) { reject; }
}
```

Figure 2.3: Indus program for stateful firewall.

instead of all packets going through a choke point. Following standard techniques for ensuring control plane consistency, the control plane could add firewall rules to all edge switches in response to a single report [101].

The program uses the input packet's contents to check if it is allowed to reach the destination, and then carries the flag in the packet. At each hop, the program checks if a packet along the reverse direction has been seen (in the `telemetry` block), and if not, the program generates a report containing the IP addresses so that the control plane can install the corresponding rules in the `allowed` dictionary. Dictionaries are implemented using P4 tables, as described in Section 2.4.

In the `init` block, which executes when a packet enters the network at the first hop, if the source and destination IP address tuple is not in the allowed dictionary (added by the control plane), then the packet is marked as violating the firewall rule and will be rejected by the checker. When a packet reaches its last hop, if the source and destination IP addresses are not in the allowed dictionary, Hydra sends a report to the control plane to add it. `last_hop` is a built-in keyword that evaluates true if and only if a packet is at its last hop before it egresses the network.

The three Hydra programs presented in this section are examples of properties that could not be fully verified by a static checker. It is possible to imagine a checker that enforces that the proper tenant isolation rules are installed, or a model checker that ensures that a switch complies with the firewall rule. But this doesn't guarantee correct runtime behavior: For example, a bug in the control-plane might install an incorrect filtering rule; or a bug in the compiler or data plane might not process a packet in the way the static checker assumed. Similarly, hardware faults (memory errors, bit flips on signals, failing connectors) would be undetectable by a static checker. Of course, Hydra programs can have bugs too. But it is less likely that the same bug would appear in the forwarding *and* the checker. This independence between forwarding and checking is key to the value of runtime verification.

p	::=	$\overline{d}\ s_{init}\ s_{tele}\ s_{check}$	*Programs*
d	::=		*Declarations*
	|	`tele` $t\ x := e$	
	|	`sensor` $t\ x := e$	
	|	`header` $t\ x$	
	|	`control` $t\ x$	
e	::=		*Expressions*
	|	x	
	|	v	
	|	$\oplus(\overline{e})$	
	|	$e_1[e_2]$	
s	::=		*Statements*
	|	`pass`	
	|	$s_1; s_2$	
	|	$x := e$	
	|	`if` (e_1) `then` s_1 `else` s_2	
	|	`for` $(x$ `in` $e)\ s$	
	|	*exn*	

t	::=		*Types*
	|	`bit`$\langle n \rangle$	
	|	*bool*	
	|	$t[n]$	
	|	`set`$\langle t \rangle$	
	|	`dict`$\langle t_1, t_2 \rangle$	
$\oplus$	::=		*Operators*
	|	$+ \mid - \mid * \mid /$	
	|	$\sim \mid \& \mid \mid$	
	|	$== \mid < \mid <= \mid ! \mid \&\& \mid \mid\mid$	
	|	$\in \mid \notin$	
	|	`length` | `push`	
v	::=		*Values*
	|	n	
	|	b	
	|	$[\overline{v}]$	
exn	::=		*Exceptions*
	|	`report`	
	|	`reject`	

Figure 2.4: Indus syntax.

2.3 The Indus Language

Having introduced some of the main features of Hydra by example, give here is a more precise definition of Indus, a domain-specific language used to specify network-wide properties. To a first approximation, an Indus program can be thought of as a classical runtime monitor that is attached to each packet traversing the network. The monitor runs alongside the forwarding code in the data plane at line rate. It can observe the behavior of each switch on the network-wide path, maintain state in telemetry variables that are carried along with the packet, and aggregate information across multiple packets using sensors.

2.3.1 Language Design

Before delving into the details of Indus, it is worth asking: why design a new language? Generally speaking, prior work on runtime verification has followed one of two approaches. The first uses formal logic to specify correctness properties. For example, to stipulate that a packet must not visit switch A twice, we could use the following formula, $\Box\ \neg(A \wedge \bigcirc(\Diamond\ A))$, which is written in Linear Temporal Logic over Finite Traces (LTL_f) [28]. Formally, it says that globally ($\Box$), it is not the case that some event satisfying A (i.e., the packet being at switch A) is followed by ($\bigcirc$) an event where A eventually occurs ($\Diamond$). More intuitively, it says that the packet must not traverse a topological loop involving switch A.

But while formal logic is very well understood, we ultimately elected not to use it as the specification language for Hydra. First, we did not believe that network operators would like or use formal logic. Second, it was not clear how to cleanly accommodate all of the state related to packet-processing—*e.g.*, packet headers and metadata, mutable state on switches, not to mention any new data we might add to support verification [29]. Instead, we followed the second main approach used in runtime verification, relying on a domain-specific instrumentation language (e.g., Eagle [14] or JavaMOP [26]) to specify correctness properties. Here, the programmer writes a program that monitors the execution of the program being verified, using introspection features such as aspect-oriented programming. Ultimately, the program implements a predicate that determines whether the execution should be allowed or not.

The design of Indus is guided by three fundamental principles. First, it provides direct access to all state in the data plane and the control plane that could be relevant to how a packet is processed. To put it another way, the language strives to make it easy to observe network-wide behaviors. Second, the language enforces a strict separation between the variables that track network state, which are read-only, and other variables, which can be read and written. This separation is to ensure that the Indus program does not interfere with the network's forwarding behavior, except at the edge, where it rejects packets that violate the specified property. Third, the language incorporates a number of restrictions to ensure that programs can be compiled to high-speed packet-processing hardware—e.g., all state must be statically allocated and it must be possible to show that all loops terminate.

2.3.2 Syntax and Semantics

Indus syntax is based on familiar imperative programming constructs (e.g., variables, conditionals, loops, etc.) and it provides a rich set of data types (e.g., bitstrings, booleans, arrays, sets, dictionaries, etc.) and operators (e.g., arithmetic, boolean, and bitwise operations) to express network-wide correctness properties. Figure 2.4 defines the formal syntax of a core fragment of the language. Our prototype implementation supports a few extensions to this core language, such as `for` loops that iterate over multiple variables, `report` exceptions that carry values, etc. These are elided from our formalization for simplicity. A program p consists of a list of declarations $\overline{d}$ followed by an initialization block, telemetry block, and checking block. Each variable is tagged with a modifier: `tele` variables reside on the packet and aggregate information along the network-wide path; `sensor` variables reside on the switch and aggregate information across multiple packets; `header` variables provide read-only access to data plane variables, such as packet headers and metadata; likewise `control` variables provide read-only access to control-plane state and other configuration information.

The initialization block is executed when the packet first enters the network. Its purpose is to perform computations that cannot be easily encoded using the initializers for variable declarations—e.g., computing a function over multiple control-plane variables. The telemetry block is executed

$$\varphi ::= A \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \bigcirc\varphi \mid \varphi_1 \,\mathcal{U}\, \varphi_2$$

$$\begin{aligned}
[\![A]\!]_x &\triangleq A(x) \\
[\![\neg\varphi]\!]_x &\triangleq \neg[\![\varphi]\!]_x \\
[\![\varphi_1 \wedge \varphi_2]\!]_x &\triangleq [\![\varphi_1]\!]_x \wedge [\![\varphi_2]\!]_x \\
[\![\bigcirc\varphi]\!]_x &\triangleq \exists y.\, succ(x, y) \wedge [\![\varphi]\!]_y \\
[\![\varphi_1 \,\mathcal{U}\, \varphi_2]\!]_x &\triangleq \exists y.\, x \leq y \wedge y \leq last \wedge [\![\varphi_2]\!]_y \wedge \\
&\qquad \forall z.\, x \leq z \wedge z < y \Rightarrow [\![\varphi_1]\!]_z
\end{aligned}$$

Figure 2.5: LTL_f syntax (top) and encoding into first-order logic (bottom) [28].

at each hop. Often the telemetry block will `push` data obtained from `header` variables into arrays maintained in `tele` variables, but other approaches are also possible. The telemetry block can also update `sensor` variables. Finally, the checking block is executed at the last hop. Its main purpose is to decide whether the packet is allowed to exit the network or if it needs to be rejected and/or reported to the management plane.

By design, Indus is strongly typed, which means all operations are checked to ensure that variables are used in ways consistent with their declaration. Types are also important for ensuring termination — e.g., because arrays have a maximum size that is known at compile time, `for` loops are guaranteed to terminate. As mentioned above, the language also enforces a clear separation between data-plane and control-plane variables, which are read only, and telemetry, sensor, and local variables, which can be read and written. Formally, this ensures that for packets that do not trigger a property violation (i.e., by raising an exception), the final output packet(s) will be identical to the packet(s) that would have been produced had the Indus program not been running at all. To put it another way, Indus does not interfere with the execution of packets that satisfy the property, only those that violate it. Similarly, the telemetry, sensor, and local variables, which are used to implement the checking logic, are kept separate from the other variables. Hence, the network cannot subvert the property being enforced simply by injecting certain packets into the network or issuing control-plane commands. Indus can be used to verify that a network is free of infinite forwarding loops, but the overhead is non-trivial—one must either enforce a maximum length on forwarding paths, or keep track of the packet's path and periodically check for duplicates. Moreover, because loops are almost always undesirable, many networks already offer robust mechanisms for avoiding them—e.g., IPv4's time-to-live (TTL) field. Hence, in examples, we will often elide the additional logic that would be needed to encode loop freedom in Indus.

2.3.3 Expressiveness

Having defined Indus, it is natural to wonder about the class of properties that it can capture. Generally speaking, questions about expressiveness are settled by giving translations that map programs from one language into another—e.g., this is how we show that Turing machines and

λ-calculus capture the same class of computations. We are not aware of any logic or existing language that precisely captures the set of properties that can be expressed using an Indus program. Among other things, the presence of `header` and `control` variables, which operate as a kind of "foreign function interface" to the data plane and the control plane, as well as `sensor` variables, make the relationship difficult to state. Nevertheless, to establish a lower bound, it is proved here that Indus is rich enough to express all LTL$_f$ formulas. Along the way, it is shown that Indus also corresponds to first-order logic formulas over finite traces.

Recall that LTL$_f$ can be understood as defining predicates on traces. Each trace is made up of an ordered sequence of events, which are assumed to be finite. Figure 2.5 gives the formal syntax of LTL$_f$. Formulas A correspond to atomic predicates that either hold or do not hold at a given event. For instance, atomic predicates could keep track of the location of the packet in the network, or the value of the destination address in the IPv4 header. Formulas $\neg\varphi$ and $\varphi_1 \wedge \varphi_2$ correspond to logical negation and conjunction respectively. Formulas $\circ\varphi$ state that φ holds in the *next* event—i.e., the one that follows the current event in the ordered sequence. Finally, formulas $\varphi_1 \,\mathcal{U}\, \varphi_2$ state that φ_1 holds at all events *until* some point at which φ_2 holds. As usual, other formulas can be encoded. For example both $\Box\varphi$, which states that φ always holds, and $\Diamond\varphi$, which states that φ eventually holds, can be encoded using the until operator.

In their original paper on LTL$_f$, De Giacamo and Vardi proved that formulas can be translated to first-order logic [28] over finite sequences. The bottom half of Figure 2.5 gives the translation, which is parameterized on a variable x corresponding to an index in the sequence, initially the index of the first element. Hence, to prove that TPC can express the same set of properties as LTL$_f$, we simply have to show that it can model the semantics of these first-order formulas. Assume that the telemetry block populates an array T with an increasing sequence of integers as well as arrays A corresponding to the atomic predicates occurring in the program. With this encoding, it is straightforward to show that the semantics of every first-order formula used in the translation of LTL$_f$ can be expressed in Indus. For example, existential formulas $\exists x.\, P$ map to a `for` loop:

```
bool r_0 := false;
for (i in T){
  x := i;
  (|P|)_{r_0};
  r := r || r_0;
}
```

Here $(\!|P|\!)_{r0}$ denotes the translation of P using an auxiliary variable r_0 to store the result. For the complete formalization, please see the technical report [104].

Theorem 1 (Expressiveness)**.** *Let φ be an LTL$_f$ formula, π a trace, $\mathcal{I}$ a corresponding first-order interpretation, and σ the corresponding Indus store. Also let $P = [\![\varphi]\!]_x[x \mapsto 1]$ and $s = (\!|P|\!)_r$. The*

```
// Hydra Headers
struct hydra_header_t {
  eth_type2_t hydra_eth_type;
  bit<8> tenant;
}
struct hydra_metadata_t {
  bit<8> tenant;
  bool reject0;
}
// Generated Init Code
apply {
  ...
  // look up ingress port tenant
  tenants_in_port.apply();
  // initialize tele variable
  hydra_header.tenant = hydra_metadata.tenant;
}
...
// Generated Checker Code
apply {
  // lookup output port
  tenants_eg_port.apply();
  if (hydra_header.tenant!=hydra_metadata.tenant) {
    // reject if ingress and egress disagree
    hydra_metadata.reject0 = true;
  }
  strip_telemetry(); // strip telemetry at last hop
}
```

Figure 2.6: Generated `tna` code for bare-metal multitenancy.

following are equivalent.

- $\pi, i \models \varphi$
- $\mathcal{I} \models P$
- $\langle \sigma, s \rangle \Downarrow \langle \sigma', s' \rangle$ *and* $\sigma'(r) = \texttt{true}$.

Proof. The first two cases were given by De Giacamo and Vardi [28]; the third case follows by induction. □

Corollary 1. *Every network-wide property that can be expressed in LTL_f can be expressed in Indus.*

Overall, this result shows that Indus is at least as expressive as the specification language used in many other runtime verification systems, modulo the choice of atomic predicates A.

2.4 The Indus Compiler

This section presents our compiler, which converts Indus programs to P4 code, which can then be linked with the forwarding code. Our compiler is designed to make it easy to ensure that the state and control-flow of the Indus program are not tampered with during this process. Our compiler is written in approximately 2500 lines of OCaml code. Our current prototype only supports P4, but it would be possible to extend it to target other DSLs for forwarding behavior, including eBPF and DPDK.

2.4.1 Code Generation

The compiler takes as inputs an Indus program and a topology file in which each switch is classified as an edge or non-edge switch. The compiler then generates switch-specific code for each switch in the topology.

The front-end of the compiler first lexes and parses the Indus program into an abstract syntax tree. Next, the type checker ensures programs are well typed and respect constraints such as read-only access to `control` and `header variables`. The type checker also constructs a symbol table for the declarations in the Indus program, which is used in the construction of the P4 headers and parsers. Finally, the compiler generates P4 code for each Indus construct. Many of the abstractions found in Indus can be directly mapped onto analogous constructs in P4—e.g., assignments, conditionals, etc. But for some other abstractions, it is not obvious how to implement them in P4. The following list summarizes the strategies used to generate code in the compiler:

- `header` variables: A header variable declaration requires an annotation (indicated with an `@`) that specifies the corresponding name in the forwarding program that tells the compiler how to translate references to the variable. For example, if an Indus program needs to refer to the source IP address through the `ip_src` variable, the required annotation is `hdr.ipv4.src_addr`. In examples, we omit these annotations for brevity.

- `tele` variables: A telemetry variable declaration leads to an extra field in a special telemetry header generated by the compiler. The `tele` variables travel with the packet as telemetry and are serialized and deserialized using parsers and deparsers generated by the compiler.

- `sensor` variables: A sensor variable declaration is implemented as a P4 register. Reads and writes to sensor variables are translated, provided the underlying target (e.g., BMv2, Tofino) supports them.

- `control` variables: A control variable declaration is mapped to a match-action table. There are two different types of control variables: a non-dictionary control variable and a dictionary control variable. A non-dictionary control variable is statically defined by the control plane, and can be initialized by a default action in a single match-action table that executes at the start of the pipeline. On the other hand, a dictionary control variable requires more complex lookups. To ensure the lookup operation returns the most up-to-date value for each dictionary control variable, our compiler creates and places a match-action table right before the statement that contains the lookup in the translated P4 code.

- Lists and loop operations: Lists are implemented as header stacks in P4, which have the semantics of a fixed length array. P4 does not support loops. Thus, our compiler *unrolls* Indus's `for` loops into sequential code: the loop body is executed for each list index that is valid. Our

compiler also supports the `in` operator, which translates into an expression that tests if the left-hand side is equal to any valid elements of the header stack specified on the right-hand side.

At the final hop, before a packet exists the network, we strip the checking headers produced by the Indus program. This ensures conformance with software running on end hosts that do not recognize the extra headers injected by Indus. To this end, the control plane needs to specify the set of edge ports in the network to the compiler. Then the compiler generates an extra match-action table that matches on the egress port and strips the headers for packets that are sent to these egress ports. A similar process is done for injecting Indus-generated headers to packets at the first hop. In principle, we could delegate these "last-hop" and "first-hop" tasks to the NIC at end hosts. We leave this extension to future work.

2.4.2 Linking

Figure 2.6 shows the generated P4 code for the bare-metal multi-tenancy example described in Section 2.2. The final compilation step is to link the generated headers and parsers blocks as well as the `init`, `telemetry`, and `checker` code blocks with the forwarding code for the switch, which we assume is also written in P4. Specifically, the `init` block must be placed at the beginning of the ingress pipeline on first-hop switches, the `telemetry` block is placed at the egress pipeline on every switch, and the `checker` block is placed at the end of the egress pipeline on last-hop switches. Since networks are bidirectional, the edge switches in the network end up running all three code blocks, while the non-edge switches only run the `telemetry` block. Automatically linking our compiler output blocks with the forwarding P4 program is future work.

2.4.3 Last-Hop vs. Per-Hop Checking

Our current compiler compiles Indus programs to the network so that a switch at every hop collects telemetry but the check only runs at the last hop, or edge, switch. This approach has a number of advantages. First, it saves resource usage on non-edge switches since running a check at a switch requires additional computation. This approach is also more amenable to incremental deployment since Hydra can still run with switches that are not fully programmable but can run telemetry and attach information to packets. Another approach, however, is to execute checks at every hop. The main advantages of this approach are that it often requires less telemetry data, and packets that violate the given property can be rejected (or reported or tagged) at any switch, not just at the edge. We plan to implement this approach in the future, using our compiler to automatically relocate checks from the edge and into the network core.

```
control bool is_spine_switch;
tele bool visited_spine;
tele bool to_reject;

init {
  visited_spine = false;
  to_reject = false;
}
telemetry {
  if (is_spine_switch) {
    if (visited_spine) {
        to_reject = true;
    }
    visited_spine = true;
  }
}
checker {
  if (to_reject) {
    reject;
  }
}
```

Figure 2.7: Valley-free routing in Indus.

2.5 Hydra Case Studies

This section presents a pair of case studies that demonstrate the practical utility of using Hydra for enforcing network-wide properties using runtime verification. The first case study develops an application of Hydra to implement path validation in a data center network with source routing, ensuring that packets follow "valley-free" paths. The second case study illustrates a use of Hydra to detect a subtle bug in Aether's implementation of application filtering, which provides a form of slicing.

2.5.1 Example 1: Valley-Free Source Routing

Recall that in source routing, the sender specifies the path the packet should take through the network. In its purest form, the path is specified as a list of hops, and each switch simply pops the stack and forwards the packet accordingly. Source routing has many advantages—e.g., it eliminates the need for large routing tables and complex routing protocols, since senders are responsible for computing paths. One downside, however, is that source routing does not offer operators the same degree of control as traditional, destination-based forwarding schemes. With Hydra, operators can specify and enforce policies that restrict the set of legal paths when using source routing; any packet that attempts to follow an illegal path will be automatically dropped. For example, an important property in data center routing is that paths are valley-free, preventing an explosion of suboptimal paths in a fat-tree topology. In particular, packets may not traverse an link that goes "up" in the topology after they have already traversed a "down" link.

Indus checker for source routing. Figure 2.8 depicts the topology of the simple network we instrumented with Hydra, generalizing code found in the P4 Tutorial [27]. The network contains a

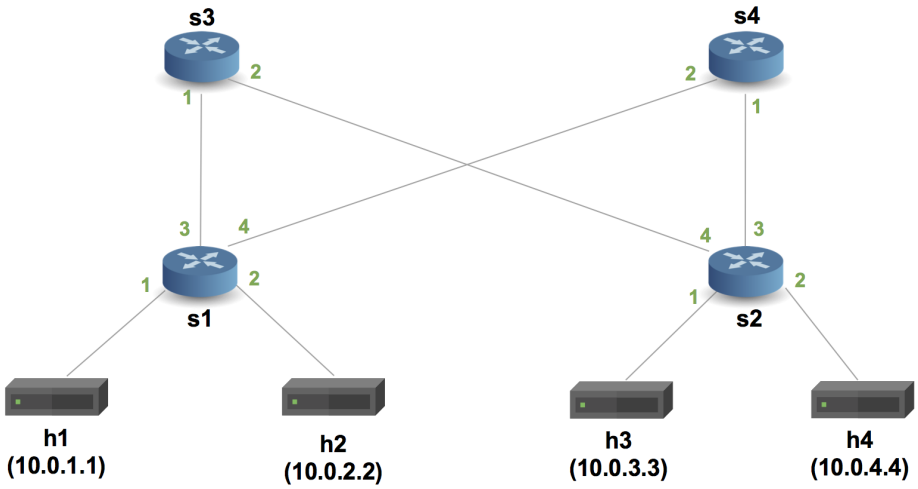


Figure 2.8: Simple Leaf-Spine Topology

leaf-spine topology with four switches. All the switches run the same P4 program, which implements a simple source routing scheme, and we link the program with the valley-free routing checker written with Indus, shown in Figure 2.7. While it is possible to write a general Indus program to check valley-free routing for any given fat-tree topology, we leverage the fact that Indus is topology-specific to write an efficient program that only requires a single control variable and two bits of telemetry to ensure that a spine switch is visited at most once. This program consists of a simple state machine that checks if the current switch is a spine switch and marks the packet to be dropped if it has already visited a spine switch. Note that the Indus program is independent of the forwarding P4 code: it could operate on any routing protocol. And while the forwarding program operates on egress ports, the Indus program operates at a higher level, using switch-specific control plane state.

Bug caught by Hydra. In this case study, we artificially injected a bug into the script used by the sender to add extra invalid hops to the source route. Using Mininet [67], we generated a number of paths and verified that Hydra allowed all possible valley free paths between hosts and successfully dropped any packets that followed errant paths due to the bug in the sender script.

2.5.2 Example 2: Application Filtering in Aether

Aether [35] is an open-source edge computing platform that offers private LTE/5G connectivity. Figure 2.10 shows an Aether edge deployment with three main elements: (1) small cells that provide LTE or 5G access to mobile clients such as cameras, sensors, or phones; (2) servers that run edge-applications exposing low-RTT services to mobile clients; and (3) an SDN fabric of P4-programmable switches that connects small cells to servers and the Internet [36].

The Aether software stack includes an operator-facing portal and API for system configuration, a 3GPP-compliant dual-mode 4G/5G mobile core, and ONOS, a distributed SDN controller responsible

```
tele bit<32> ue_ipv4_addr;
tele bit<32> app_ipv4_addr;
tele bit<8> app_ip_proto;
tele bit<16> app_l4_port;
tele bit<8> filtering_action = 0; // 1=deny,2=allow
control dict<(bit<32>,bit<8>,bit<32>,bit<16),bit<8>> filtering_actions;

header bit<32> inner_ipv4_src;
/* ... Header variable declarations ... */
header bit<16> outer_udp_dport;

init {
  if (inner_ipv4_is_valid) {
  // this is an uplink packet
    ue_ipv4_addr = inner_ipv4_src;
    app_ip_proto = inner_ipv4_proto;
    app_ipv4_addr = inner_ipv4_dst;
    if (inner_tcp_is_valid) {
      app_l4_port = inner_tcp_dport;
    } elsif (inner_udp_is_valid) {
      app_l4_port = inner_udp_dport;
    }
  } elsif (ipv4_is_valid) {
  // this is a downlink packet
    ue_ipv4_addr = outer_ipv4_dst;
    app_ip_proto = outer_ipv4_proto;
    app_ipv4_addr = outer_ipv4_src;
    if (tcp_is_valid) {
      app_l4_port = outer_tcp_sport;
    } elsif (udp_is_valid) {
      app_l4_port = outer_udp_sport;
    }
  }
  filtering_action = filtering_actions[(ue_ipv4_addr, app_ip_proto, app_ipv4_addr, app_l4_port)];
}
telemetry {}
checker {
  if (filtering_action == 1 && !to_be_dropped) {
    reject; report((ue_ipv4_addr, app_ip_proto, app_ipv4_addr, app_l4_port, filtering_action));
  }
  if (filtering_action == 2 && to_be_dropped) {
    report((ue_ipv4_addr, app_ip_proto, app_ipv4_addr, app_l4_port, filtering_action));
  }
}
```

Figure 2.9: Aether application filtering in Indus.

for controlling the fabric switches. The fabric provides L3 connectivity by routing IPv4 packets over the spine switches using Equal Cost Multi-Path (ECMP) forwarding. It supports L2 bridging and VLAN isolation within a rack, and other common features such as rerouting in case of failures, learning/advertising routes via BGP, configuring static-routes, DHCP relay, multicast, and ACLs for filtering. A notable feature in Aether is that the switches help implement the mobile core User Plane Function (UPF) [75] (i.e., with support for GTP-tunnel encapsulation/decapsulation, downlink buffering, accounting, QoS, application-filtering, and slicing).

Aether application filtering. We implemented a wide range of Hydra checkers for Aether (see Section 2.6, Table 2.1), but we focus here on UPF application filtering, which had a subtle bug we detected using Hydra. Application filtering allows operators to create slices that connect an isolated group of clients and give them with bandwidth guarantees. Operators can define filtering rules allowing clients in a slice to access some edge-applications while denying access to others. Internet

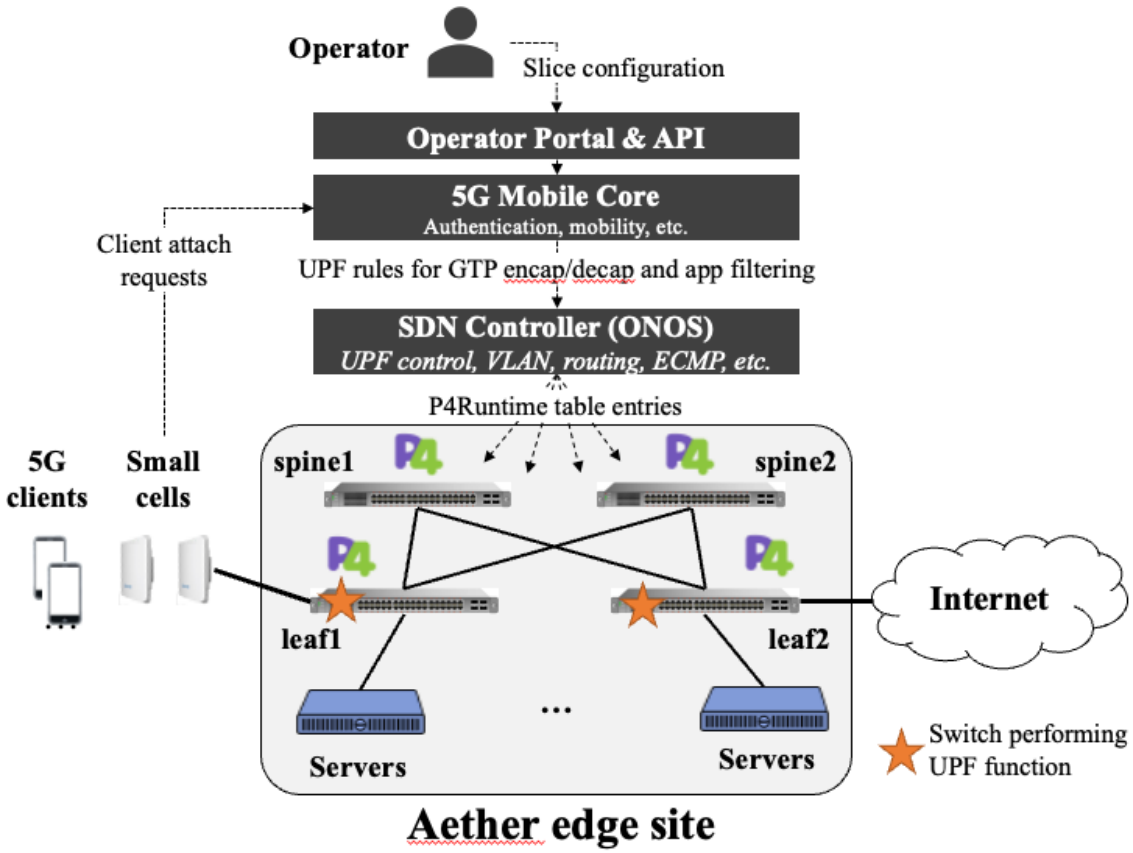


Figure 2.10: Aether architecture and topology.

access is considered an application, and applications can be shared across slices. For example, mobile clients belonging to the camera-slice are allowed to communicate with an edge application that analyzes video, but cannot access the Internet. Mobile clients in the phone-slice have the opposite permissions.

Each slice has a prioritized list of filtering rules of the form:

`priority` : `ip-prefix` : `ip-proto` : `l4-port` : `action`

where `ip-prefix`, `ip-port`, and `l4-port` identify the application. The `action` can be `allow` or `deny`, and the priority is used to disambiguate in case of overlapping rules. For example, to deny all traffic by default but allow access to applications using UDP port 81, the operator could use the following rules:

- `20 : 0.0.0.0/0 : UDP : 81 : allow`
- `10 : 0.0.0.0/0 : any : any : deny`

Now, to integrate with any 3GPP-compliant mobile core, Aether's ONOS controller uses a standard 3GPP interface named PFCP. This interface does not allow to specify application filtering rules globally for a slice. Instead, rules are sent to ONOS on a per-client basis. This means that ONOS receives the same application filtering rules for every client that connects to the network. Thus, in each slice, there are a set of clients (identified by their IMSI, a unique number associated with a SIM card) and a list of application filtering rules for each client. When a new client connects to Aether, the mobile core looks up the slice configuration for the given IMSI and installs the user plane rules on switches to terminate the GTP tunnels and enforce application filtering. The P4 program running on the switch optimizes ASIC resources by splitting UPF processing across different types of tables, and ONOS is responsible for translating UPF rules into multiple table entries and updating the entries in each leaf switch. Hence, while the slice and application filtering configuration is conceptually simple, ensuring the correctness of the filtering depends on the interaction of multiple software and hardware components, each of which could be subject to different bugs. Bugs and errors could result in the installation of erroneous entries, which may cause traffic to violate the intended policy.

Hydra checker for application filtering. Figure 2.9 shows the Indus program to verify application filtering. The `init` block first determines the direction of the packet and then fetches the fields of interest into `tele` variables, which it then uses to look up a `control` variable to know the filtering action. The filtering action is carried on the packet (in addition to the packet fields used in the lookup). A simple control plane application that runs atop ONOS as part of the rest of the deployment configures the `control` dictionary variable. At a high level, it receives the application filtering rules from the operator at startup, listens for attach requests from mobile clients, and installs the corresponding entries in the table for the `filtering_actions` variable.

Bug caught by Hydra. We now describe a known bug in Aether that causes traffic to be dropped when updating application filtering. Figure 2.11 provides a simplified representation of the multiple P4 tables used to realize the UPF function. To reduce memory utilization (in particular, of TCAM), the *Applications* table is designed so that entries can be shared by multiple clients of the same slice. This table determines the application for a packet by matching on the IPv4 and L4 port headers, and sets the appropriate app ID metadata for the packet. The *Terminations* table then uses the app and client IDs together to determine whether to forward or drop the packet. This design requires ONOS to correctly manage shared application entries when clients connect to the network or when rules are updated in the operator portal.

Figure 2.11 shows a scenario where a specific slice is first configured with filtering rules that deny all traffic by default (app ID 1) but allow traffic for apps on UDP port 81 (app ID 2), which has a higher priority. When client ID 1 connects, two rules are installed: the default drop rule for client ID 1 with app ID 1, and the allow rule for client ID 1 with app ID 2. Thus, client ID 1 can successfully access applications on UDP port 81. Let's say the operator later updates the filtering rules in the

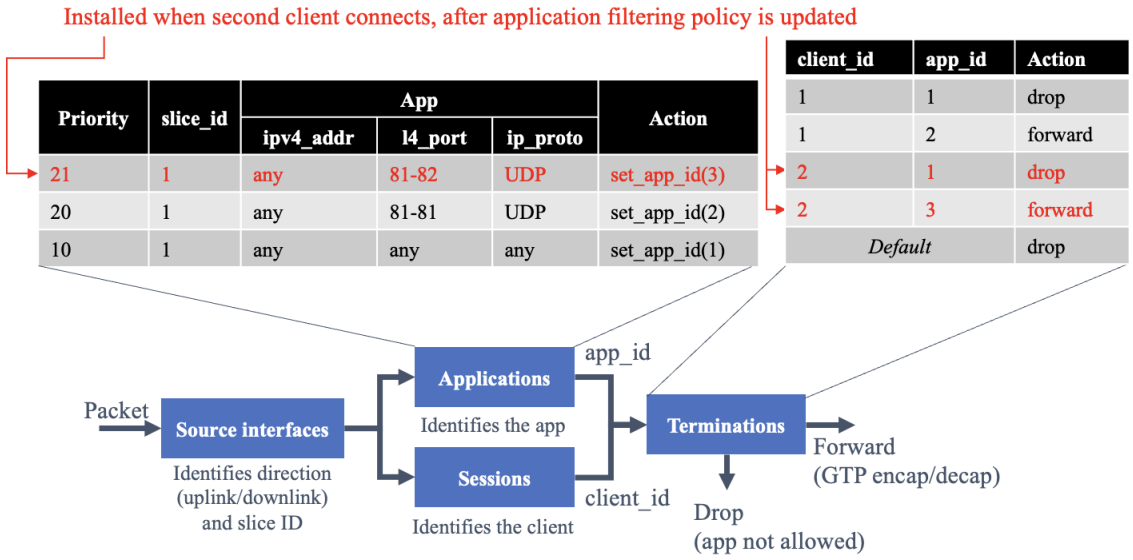


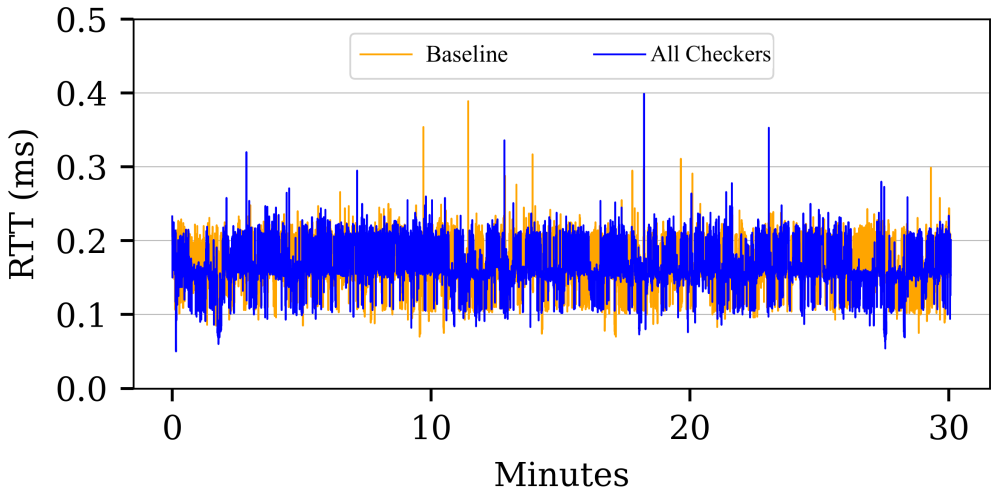
Figure 2.11: P4 tables demonstrating application filtering bug

portal by expanding the UDP port range to 81-82 and increasing the priority of that rule, and set the app ID as 3. When client ID 2 connects, the mobile core installs client-specific rules with this updated policy, thus ONOS installs a new table entry with range 81-82 in the *Applications* table. Due to the new higher-priority entry for app ID 3, packets from client ID 1 with UDP port 81 will now get an app ID 3 assigned by the *Applications* table. As a result, traffic for client ID 1 on port 81 that was previously allowed is now dropped since the client-app ID pair does not exist in the *Terminations* table. This subtle bug is hard to catch and even harder to pinpoint the exact location where the packets are being dropped.

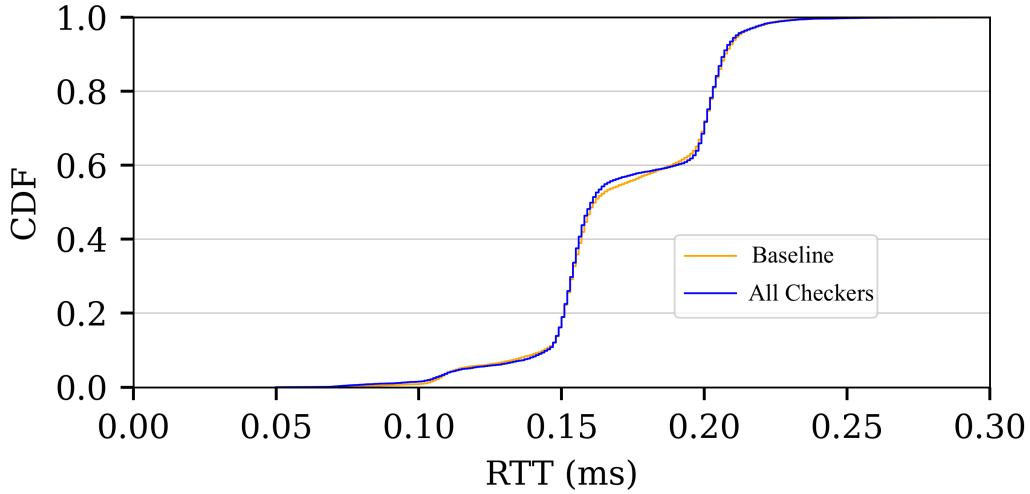
With the checker compiled from Figure 2.9, Hydra detects that client ID 1's packets with UDP port 81 are actually to be dropped when it should have been allowed. With the `report` action, such behavior is explicitly reported to the control plane by the switch where the inconsistency was detected.

2.6 Evaluation

To further evaluate our design, we wrote more checkers for verifying a range of properties in the Aether testbed, including examples studied previously in the network verification literature. We assess the expressiveness of Indus and and overheads of our Hydra system. Table 2.1 summarizes our results.



(a) RTT comparison over time



(b) RTT CDF comparison

Figure 2.12: Performance overhead of Hydra

Property Name	**Description**	**LoC**		**Tofino Overhead**	
		Indus	P4 Output	Stages	PHV (%)
Baseline	Aether P4 program compiled in `fabric-upf` profile	-	-	12	44.53
Multi-Tenancy	All traffic through a given ToR switch port, facing a bare-metal server should belong to the same tenant	14	102	11	48.44
Datacenter uplink load balance	Uplink ports in data center switches should load balance, to exact equivalence, between specified ports	37	194	12	48.83
Stateful firewall	Flows can only enter the network if a device inside initiated the communication	23	164	12	49.21
Application filtering	Clients should only be able to communicate with designated applications (as identified by layer 4 ports)	64	126	12	52.14
VLAN isolation	Packets should traverse switches in the same VLAN	21	119	11	47.85
Egress port validity	Packets should only egress a switch at allowed ports	18	132	12	46.09
Routing validity	The first and last hop of any packet should be a leaf switch, while the rest of the hops are spine switches	21	122	12	46.09
Loops (4 hops)	Packets should not visit the same switch twice	20	156	12	48.24
Waypointing	All packets should pass through a choke point	22	154	12	47.85
Service chains	Packets from switch s to switch t should pass through switches $(w_1, w_2, ..., w_n)$ in that order on the way	26	121	12	47.26
Source routing with path validation	A packet that is source routed through switches $(s, s_1, ..., t)$ should pass them in order	34	211	12	51.56

Table 2.1: Hydra properties.

2.6.1 Expressiveness and Conciseness

In Table 2.1, we show the number of lines of Indus code required to specify each property and the number of lines of P4 code generated by our compiler. Indus enables expressing properties succinctly, typically requiring an order of magnitude less code compared to the direct implementation in P4. We optimized the programs to streamline their compilation to hardware. For example, in the Indus checker for detecting load imbalance, we maintain a boolean variable that records whether an imbalance has been detected on any switch on the network-wide path, which eliminates the need to iterate over multiple arrays in the block. Overall, our evaluation shows that Indus, our domain-specific language, can express a wide range of practical network properties in a concise manner.

2.6.2 Resource and Performance Overheads

Next, we discuss the overheads associated with deploying Hydra checkers on Intel Tofino switches.
Resource Overhead. The main resources on Tofino switches that are relevant to Hydra are the number of pipeline stages used and the amount of Packet Header Vector (PHV) bits used. Other stage resources (e.g., SRAMs, TCAMs, etc.) are also important, but their contribution is implicitly accounted for in the usage of pipeline stages. We first measure the resource utilization of the baseline forwarding program that runs in the Aether mobile core and then measure the resource utilization for each of the implemented properties when linked with this program.

The baseline program is already at 12 stages. In general, deploying a Hydra checker will require extra resources. However, in this instance, each of the checkers can be executed in parallel alongside the base program and they do not increase the number of stages when linked with the base program. This parallel execution is made possible by the independence between the forwarding and checking code. We can see that the overhead on PHV resources is relatively modest, with higher usage for the programs that collect more telemetry. For instance, the properties that require the most PHV are source routing path validation and application filtering. The former carries a significant chunk of telemetry per hop while the latter collects all its telemetry in the `init` codeblock. The PHV resource usage increases from 44.53% to 52.14% with the application filtering checker on, a 7.6% difference.
Performance Overhead: Setup. Next, we evaluate if Hydra introduces any performance overhead when deployed in practice with our Aether testbed. We confirmed that mobile devices connected to the cellular network had stable Internet connectivity even when Hydra checkers were running. However, although Aether processes real-world traffic, the data rates in our testbed are currently not high enough to fully evaluate Hydra's performance limits. Thus, for this evaluation, we tapped and mirrored network traffic from a production campus network and replayed it towards `leaf1` in Figure 2.10. As illustrated in Figure 2.13, we utilize an existing P4Campus infrastructure [62] at Princeton University. The campus network has network Test Access Point (TAP) devices installed at several vantage points in the network, which create a mirror of the traffic they see on links. We tap

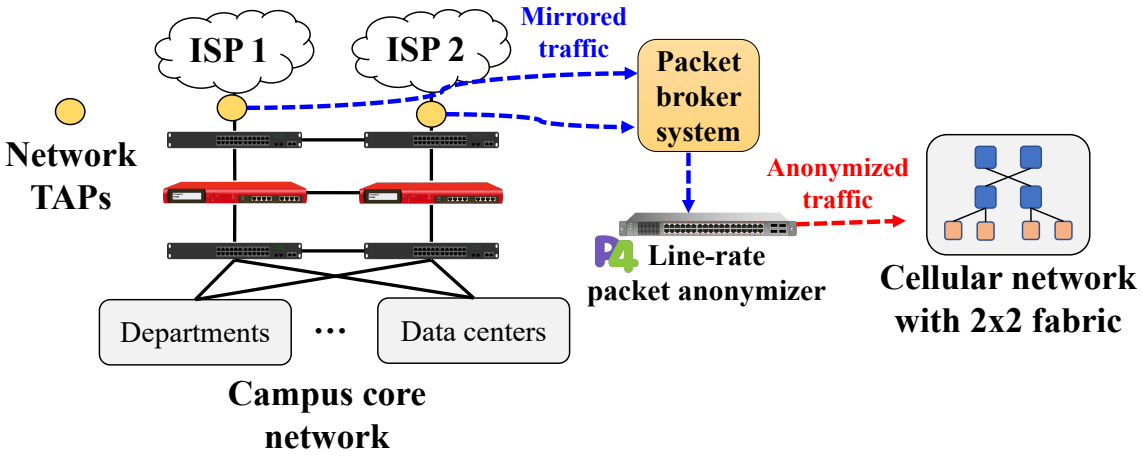


Figure 2.13: TAP architecture for mirroring campus network traffic to Aether.

two /16 campus network subnets at our border routers and send the mirrored traffic to a P4-based packet anonymizer [63], which runs on a programmable switch. This P4 program hashes personally identifiable information like MAC and IP addresses in a prefix-preserving manner at line rate and delivers the anonymized traffic to the cellular network. The resulting anonymized packet trace's load is around 350K packets per second.

Ethical considerations: All packet traces were inspected and sanitized by a campus network operator. Personal data, like MAC and IP addresses, were removed or hashed before being accessed by researchers. Addresses were anonymized in a consistent manner using a one-way hash with a salt, and payloads are discarded. Our research was discussed and approved by Princeton's Institutional Review Board (IRB).

Performance Overhead: Result. Next, we evaluate if Hydra adds noticeable performance overhead due to its parsing and checking logic. We perform a microbenchmark with and without Hydra enabled and compare the two. Our throughput comparison with and without Hydra were almost identical with around 20 Gb/s. However, we were not able to push near to the throughput limit of the hardware switches, which is is 6.5 TB/s. Thus, we focused our performance evaluation to measuring Hydra's overhead on packet-processing latency. First, we generate bidirectional UDP traffic at 10 Gb/s on our cellular access network testbed using `iperf3`. The background traffic utilizes all links between the two spine and two leaf switches with ECMP routing. Then, we started a fast ping (every 0.2 s) from one server attached to the `leaf1` switch to another service attached to the `leaf2` switch. Figure 2.12a shows the round-trip times (RTTs) during the experiment. There appears to be no significant difference between the baseline and with all checkers enabled. To further evaluate the latency overhead statistically, we plotted the cumulative distribution function of our RTT measurements in Figure 2.12b, and also performed a t-test [44]. Both results confirm that there is no statistical latency difference between the baseline and with all checkers on.

2.7 Related Work

Hydra builds on and extends a rich body of work on network verification and runtime monitoring. This section discusses the most relevant prior work, organized by topic area.

Runtime Verification. In runtime verification (RV), a system is instrumented to send events about its execution to a monitor. While the system executes, the monitor verifies the behavior against a specification. When a behavior violation occurs, the monitor sends feedback to correct the behavior or halt the execution. Over time, researchers have created more expressive languages to specify properties in runtime verification systems. Work such as Eagle [14] helped popularize the use of Linear Temporal Logic to specify properties in the monitor. Eagle's powerful logic can express a diverse set of runtime behaviors, such as requiring each request in an application to have a corresponding response or limiting the size of a queue. Eagle also provides significant flexibility in what it monitors: any system can be instrumented to send a log of events to Eagle as the structure and content of the logs are user defined.

Static Verification for Networks. There is also a large body of work on static verification for networks. Early work by Xie et al. [134] proposed using static techniques to reason about reachability in IP networks. It proposed the now-standard approach of computing the transitive closure of transfer functions that model the behavior of individual devices and links. Header Space Analysis [59], Anteater [76], and Veriflow [61] emerged later, and applied this general approach in the context SDN. To improve the scalability of their analyses, they developed optimized data structures for transfer functions. Atomic predicates [137] replaces complex classifiers with simple predicates that can be handled efficiently in backend solvers. NetKAT [8] is an algebraic framework based on a sound and complete deductive system and a decision procedure based on automata [34]. Tools like `p4v` [73], Acquila [122], and Network Optimized Datalog (NoD) [74] translate P4 code into representations that can be verified using staic techniques. Vera [116] and P4-Assert [38] address the same problems using symbolic execution, while `bf4` [30] infers control-plane constraints automatically. Gravel [142] formally verifies the software of middle boxes such as NATs and firewalls. A complementary line of work focuses on control plane verification. Batfish [33] and Minesweeper [15] statically analyze configuration files for distributed protocols to verify reachability properties automatically. Recent approaches use abstract interpretation to verify simpler representations of programs [31, 42, 16]. Our work builds on the extensive foundation provided by this prior work, but uses an approach based on runtime rather than static techniques.

Runtime Verification for Networks. Early work on runtime checking for networks focused on generating test or probe packets [140, 83, 23]. P4Consist proposes adding a new module to tag packets with the path and forwarding rules used to process them [112]. However, P4Consist only adds these tags to special probe packets, which are generated using a traffic generator, and verification is performed out-of-band—i.e., the tagged packets are sent to a separate server for analysis. VeriDP follows a similar approach, but focuses on detecting inconsistencies between control-plane

and data-plane state [143]. In contrast, Hydra instead uses checkers that execute directly in the data plane, allowing it to detect violations as they occur and halt erroneous packets. Offline analysis approaches also face inherent scalability issues or accuracy tradeoffs due to sampling. Hydra does not rely on sampling—it performs runtime checking on every packet. DBVal verifies assertions at runtime by instrumenting the data plane [66]. However, their checks are tied to how forwarding is implemented. Thus, the verification code and system being instrumented are not independent, which could lead to false negatives if forwarding and checking code have the same bug. Aragog [138] supports defining properties parameterized on location, stateful variables, and temporal predicates. Additionally, it checks every execution trace in the system. Aragog differs from Hydra in that it focuses on distributed network functions, rather than the data plane itself. In that sense, it is a complementary approach to Hydra. Aragog requires making modest modifications to the source code for the network functions, in order to send events of interest to the verifier. Hydra checks every packet and is independent of the forwarding code.

Summary. Hydra builds on ideas developed in the runtime verification and formal methods communities and applies them to the problem of verifying network behavior. It provides a specification language for expressing network-wide properties and a compiler that translates these programs into executable code for network switches. The approach is expressive, scalable, and operates in-band, detecting and blocking errant packets at line rate and in real time.

2.8 Discussion and Summary

This chapter presented Hydra, a system for runtime verification of network behavior. The key insight is that there is an important difference between catching a packet on the wrong path *immediately* versus catching it *eventually*. If an intruder is exfiltrating confidential data, one packet may be all it takes; if a single packet passes between two "isolated" virtual tenants, trust (and business) is lost. Hydra's approach is to check *every* packet as it flows through the network. While this is an extreme approach, it becomes essential as we move toward automated closed-loop control of networks with minimal human intervention.

Hydra programs expressed in Indus are easy to read and write for a large set of expressive properties. Our experiences deploying Hydra programs on P4 switches and in the context of an open-source cellular access network with real-world traffic demonstrate that they create little overhead, yet can catch real bugs in a live system. The evaluation shows that Indus can express all properties encodable in Linear Temporal Logic over finite traces (LTL_f), and that the runtime overhead is modest whether measured in pipeline resources, packet-processing latency, or throughput.

Our Hydra examples demonstrate that leveraging application-specific knowledge—in this case, knowledge of desired network properties—enables practical runtime verification at line rate, complementing the static verification techniques discussed in Chapter 1.

Chapter 3

Prescheduled Circuit Switching for LLM Training

3.1 Introduction

As discussed in Chapter 1, LLM training traffic is regular and predictable, with communication patterns largely determined before the training run starts [97, 69]. Despite this predictability, current training systems employ packet-switched fabrics: scale-up networks connect accelerators within racks (e.g., NVLink [89], TPU ICI [56], AMD Infinity Fabric [113], AWS NeuronLink [109]), while scale-out networks connect racks together (e.g., Infiniband [88], RoCE [41], UEC [123]). While packet switching excels at handling unpredictable traffic from heterogeneous applications, training systems rely on over-provisioned fabrics [41] to avoid congestion, which is inefficient in terms of power and cost. Even with over-provisioning, hotspots can delay training cycles [96]. The resulting unpredictable latency and throughput seems counterintuitive given the enormous investment in optimizing training performance.

In particular, we ask: *Would a circuit switched fabric, built from pre-scheduled crossbar switches, and cognizant of the traffic patterns and sequence of arrivals, allow LLM training systems to complete sooner and/or consume less power?*

Circuit switches usually add circuits one at a time, when a new call starts. This chapter investigates whether we can pre-schedule the traffic for the *entire* training run upfront, using knowledge of the traffic pattern and the topology, and hence predetermine the configuration of the crossbar switches needed to transfer data in successive, fixed-length time slots. We envisage simple switches: no packet buffers, no forwarding tables, and no packet processing. Delays are predictable and minimized. We will assume that the fixed-length time slots time-synchronize the whole system, which is therefore essentially centrally controlled, albeit with fault-tolerant mechanisms to minimize downtime when

switches, links, NICs, accelerators, CPUs and memories fail.

LLM training systems employ several parallelisms, such as data parallelism [100], tensor parallelism [111], pipeline parallelism [49, 82], sequence parallelism [64], and context parallelism [71]. Each parallelism requires communication between a specific set of accelerators in a specific order. An important exception is expert parallelism used with Mixture-of-Experts (MoE) [32, 78] where traffic patterns are input-dependent and not known until the routing layer decides which expert the tokens should be sent to. We show how to schedule MoE traffic (and any asynchronous traffic) on the fly using Birkhoff-von Neumann [18, 127] decomposition.

Our long-term goal is to design an AI fabric that maximizes the AI fabric's total capacity (in bits/s) divided by its overall power consumption. We conjecture that a pre-scheduled crossbar-based AI fabric will consume a fraction of the power of today's Ethernet and proprietary NVlink-based systems, and can reduce the network tiers by increasing switch fanout.

This chapter describes a high-level design for a circuit-switched fabric built from simple crossbar switches. We show how to derive the switch schedules for the entire run upfront, directly from the training code. We demonstrate how unpredictable traffic (in expert parallelism and control traffic) can be scheduled quickly and efficiently using on-demand circuit allocation. We argue that the system can be at least as reliable as today's packet-switched systems.

The key contributions of this chapter are threefold. First, we propose Chronos, an AI fabric design that uses only pre-scheduled crossbar circuit switches, leveraging known *a priori* training traffic patterns. Second, we develop a technique and tool (Genstack) to analyze training source code and generate a sequence of permutations to schedule the crossbar switches for the training run. Third, we show how Chronos handles unpredictable traffic (e.g., mixture-of-experts, management and control messages) by calculating circuit switch permutations on-the-fly using Birkhoff-von Neumann decomposition.

Our work is inspired by Google's Lightwave network [72] for dynamically creating clusters of TPU nodes prior to AI training, exploiting pre-determined synchronous traffic patterns. However, Chronos takes circuit switching further by replacing *every* packet switch with a circuit switch, including at the top-of-rack. Additionally, Chronos is agnostic to the switch technology (optical or electrical), supports unpredictable traffic including MoE parallelism through on-demand scheduling, and represents a design study without a physical prototype. While previous work has noted the predictability of communication patterns during LLM training [97, 69, 98, 128] and others have investigated circuit switching for conventional datacenter traffic [94, 92, 125, 13], Chronos explores using this predictability to completely pre-schedule an entire AI fabric for the duration of training.

3.2 Predictable Traffic Patterns in LLM Training

Model developers use *collectives* to implement each type of parallelism (except MoE, which we study in Section 3.7). Collectives are patterns of communication between pre-determined groups of accelerators, rather than between individual pairs [86]. It helps to think of accelerators as arranged in an n-dimensional array, where n is the number of types of parallelism and each parallelism communicates in one dimension. Collectives, such as *all-reduce*, *reduce-scatter*, and *all-gather*, are used to communicate among accelerators within a parallelism, typically using logical rings.[1] Figure 3.1 shows a logical ring among four GPUs in the same parallelism group, and the corresponding permutation matrix.

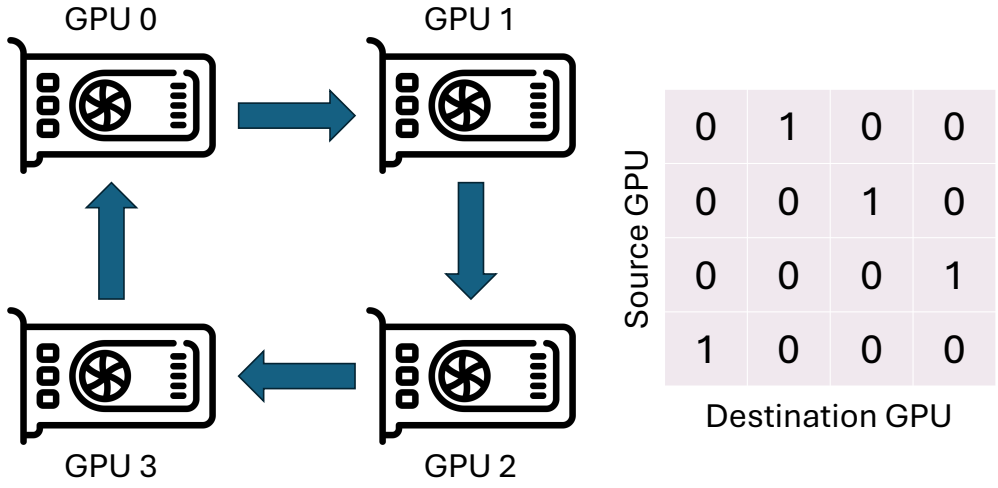


Figure 3.1: An example logical ring involving four GPUs (left) and the corresponding permutation (right).

In our approach, we determine the set (or "stack") of all permutations needed for a training run. Section 3.4 describes how we derive the permutations from the training source code. If all the accelerators are connected by one big circuit switch then the stack of permutations forms a schedule used to configure the central switch in successive rounds of communication, with one permutation per time slot. In big systems the fabric is a multistage hierarchy of circuit switches, and so we need to decompose the network-wide permutations into a local set of permutations for each switch. The decomposition is trivial if each tier has the same capacity (i.e. the fabric is not oversubscribed); we simply route all traffic via the top tier and back down again. If we want to switch locally (to reduce power, or because of oversubscription at higher tiers), we can use the method described in Section 3.5 and can potentially use shorter time slots locally than globally.

[1] Previous authors have used topology knowledge to accelerate collectives [110, 80, 21, 144, 108]. Our examples assume ring-based collectives but can in principle be extended to other implementations.

3.3 Circuit switches are simpler

A circuit switch can be built using a crossbar switch in which, during a time slot, each input is connected to at most one output and each output is connected to at most one input, as represented by the permutation matrix in Figure 3.2.[2]

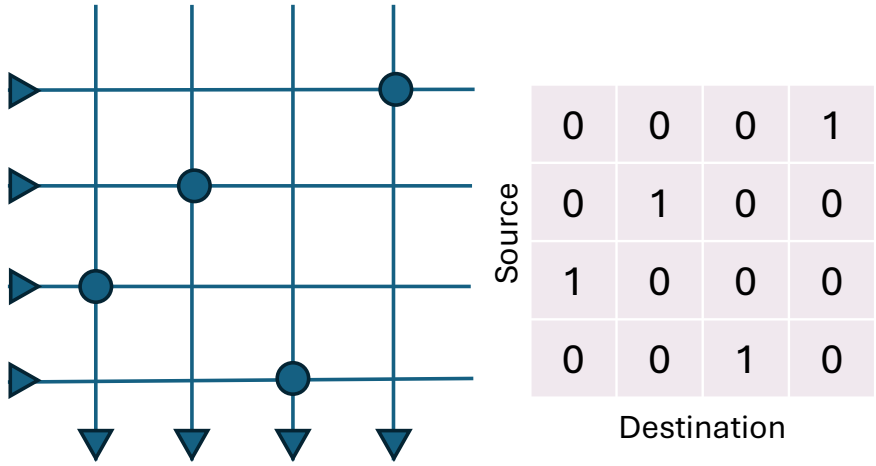


Figure 3.2: An example 4x4 crossbar switch (left) configured by a permutation matrix (right).

Circuit switches are much simpler than packet switches because they have less to do. A typical single-chip Ethernet packet switch has 30% of its area dedicated to serial I/O, 50% to packet processing and 20% to packet buffers and the traffic manager [20]. An circuit switch does not need packet processing logic, lookup tables, or packet buffers, because arriving packets are immediately switched to a pre-determined output. This makes possible electronic circuit switches with 70% of the area freed up, which means (a) lower power, or (b) we can repurpose the area to add more I/O capacity (e.g., using area I/O [135]).

Circuit switches have deterministic, low latency; a few nanoseconds for an electrical or optical circuit switch, compared to variable, unpredictable latency in packet switches.

Optical circuit switches bring additional benefits, including optical transparency and hence can carry data independent of wavelength and data rate.

3.4 Deriving permutations from training code

We wrote a software tool called Genstack to derive the switch permutations from the training code. It operates in two steps: First, it replaces collective calls with logging code and instruments how

[2]In principle, multicast and broadcast can be useful to accelerate collectives, but we don't consider them in this paper.

the collectives are called. Second, it combines collective calls into global permutations that capture communications among all accelerators. We describe the two steps in detail below. Our results are based on Megatron-LM [85], a popular and open framework for LLM training, and can be used with other LLM training frameworks [2, 3, 105].

3.4.1 Logging the collectives

Genstack logs all collective calls (e.g. `all_reduce`, `all_gather`, `broadcast`, etc.) invoked by a single iteration of training, along with their tensor shapes and participating GPUs.

But we cannot log the collective calls on the real full-scale system before we have picked the permutations, and so instead we emulate a single iteration of training running on a smaller system, without GPUs. This is still challenging because: (1) We can not manipulate the full-system high-dimensional model tensors without a large number of physical GPUs, (2) we can not spawn thousands of GPU processes, and (3) training code (such as Megatron-LM) conditionally invokes communication collectives based on pipeline stage, tensor-parallel group, and data-parallel group, requiring end-to-end code execution.

We avoid needing GPUs by using *meta tensors* [95], which are zero-overhead tensors that store only shape and dtype. Throughout the model construction, forward pass, backward pass, and optimizer step, only the shapes and dtypes are tracked, rather than actual floating-point data. Operations such as matrix multiplication and layer normalization become *no-ops* during runtime; they skip kernel execution but preserve shape propagation for the next layer. We then *monkey-patch* [37]—i.e., dynamically replace at runtime—the functions in `torch.distributed` to spoof GPU execution and record the collectives rather than executing them. We use multiprocessing to spawn one CPU process for each GPU involved. This maintains correct control-flow logic without incurring the expense of real data transfers.

With these patches in place, we run exactly one training iteration (forward + backward + optimizer step). Each process logs the communication events it experiences locally:

```
[ {"op": "broadcast", "call_id": 1, "ranks": [0,1], "shape": [1024, 4096], "dtype": "float16"},
  {"op": "all_reduce", "call_id": 2, "ranks": [0,1,2,3], "shape": [1024, 4096], "dtype": "float16", "reduce_op": "
       MIN"}, ... ]
```

This trace fully captures the GPUs involved and the tensor dimensions of each collective call in an iteration. The tool is described in more detail, with code examples, in Appendix 3.4.3.

3.4.2 Deriving permutations from the logs

To derive the stack of global permutations we combine the permutations for each collective call and for each parallelism.

It helps to study a small example. Consider 16 GPUs arranged in a logical 4×4 array, and using data parallelism (x-axis communication) and tensor parallelism (y-axis communication). We need to produce a stack of 16×16 global permutation matrices to schedule the 16×16 crossbar switch.

We will start with data parallelism and consider the rows of the 4×4 array. The rows operate identically and in parallel, calling the same collective operations at the same time, and transferring the same amount of data. Specifically, they make n collective calls $(c_1, c_2, \ldots, c_n)$ causing data transfers represented by permutations $p_1, p_2, \ldots, p_n$. This allows us to generate global permutations $P_1, P_2, \ldots, P_n$ for the n collective calls across rows where P_j is generated by combining the 4×4 permutations for each row. We repeat the process along each column for tensor parallelism.[3]

If pipeline parallelism is used, there will be a different stack of global permutations per pipeline stage in the forward and backward pass. Each permutation is incomplete because the GPUs in each pipeline stage only communicate with GPUs in the previous/next stage (for pipeline parallelism) or with other GPUs in the same stage (for all the other parallelisms).

Once we have the global permutations, we generate the switch-local permutations.

3.4.3 Implementation Details

This section provides additional implementation details of our Genstack tool used to extract permutations from training code.

Meta-Tensor Execution: Since PyTorch 1.9, users can instantiate *meta tensors* [95] via:

```
x = torch.empty(16, 1024, device='meta')
```

The tensor has no backing memory, but still participates in shape inference. We intercept all tensor creation routines (e.g., `torch.zeros`, `torch.empty`, `torch.ones`, or model parameter initializations) so that they produce meta tensors by default:

```
_orig_empty = torch.empty

def meta_empty(*shape, **kwargs):
    return _orig_empty(*shape, device='meta', **kwargs)

torch.empty = meta_empty
# Similarly for zeros, ones, etc.
```

Spoofing Distributed Processes: We emulate the full training system on a small CPU-based system in two steps. First, we spawn N Python processes (each simulating one rank). Then, each process assigns ranks in the range $[0, N-1]$ by calling

```
torch.distributed.init_process_group(backend='gloo', world_size=N, rank=LOCAL_RANK, store=...)
```

Because we only use meta tensors, no actual GPU memory is allocated. PyTorch's distributed initialization runs normally, but subsequent collective calls are intercepted.

Intercepting collective calls: All collective calls are *monkey-patched*—dynamically replaced at runtime [37]—for example:

[3]The number of global permutations is different for each parallelism.

- `torch.distributed.all_reduce`
- `torch.distributed.all_gather`
- `torch.distributed.broadcast`
- `torch.distributed.reduce_scatter`
- `torch.distributed.send/recv`
- `torch.distributed.barrier`

We store the original PyTorch functions, but replace them with wrapped versions to log the call by gathering the call type (e.g., `all_reduce`), tensor shape, dtype, operation type (e.g., `SUM`), and rank inferred from the `ProcessGroup` handle. The calls return immediately with a no-op result, for example:

```
orig_allreduce = torch.distributed.all_reduce

def wrapped_allreduce(tensor, op, group=None, async_op=False):
    # 1. Identify ranks in 'group' from a stored lookup table
    ranks = group_to_ranks[group]
    # 2. Record shape, dtype, etc.
    # reduce_op is only logged for all_reduce and reduce_scatter
    log_event({"op": "all_reduce", "call_id": get_next_call_id(), "ranks": ranks,
        "shape": list(tensor.shape), "dtype": str(tensor.dtype), "reduce_op": "SUM
        "})
    # 3. Return a no-op; skip real comm
    return
```

The wrappers capture exactly which collective calls would be issued during forward/backward passes and the optimizer step.

Process Groups and Rank Membership. Megatron-LM often creates logical subgroups (data-parallel, tensor-parallel, pipeline-parallel) via:

```
data_parallel_group = torch.distributed.new_group(ranks=dp_ranks)
```

We intercept `new_group` to record which ranks are in a group. When a collective references a group, our wrapper retrieves the associated set of ranks.

The Genstack tool is open source and available on GitHub [102].

3.5 Time Slots

The duration of a time slot, T, is determined by four attributes, (a) the smallest number of bytes, b, transferred by any of the collective communication calls during a training run, (b) the link speed, c, (c) the maximum one-way latency between endpoints, t_{max}, and (d) the 'dead' time (*aka* the 'guard' time) while a crossbar changes permutation, t_x. This is $<$ 10ns for an electronic switch or an optical switch with tunable lasers, and about 1ms for a MEMs-based optical switch. A detailed comparison of different optical circuit switching technologies is in Table 3.1. The time slot duration should be chosen so that $T \geq \frac{b}{c} + t_{max} + t_x$. The additive factor of t_x is so that the crossbars can reconfigure the permutation to the one needed for the next time slot, and the additive factor of t_{max} is so that all the crossbars are in the configuration corresponding to the one-big permutation for the current time slot irrespective of the variation in one-way latencies between endpoints.

Technology	**Port count**	**Reconfiguration latency**
Robotic [120]	1008x1008	minutes (per connection)
MEMS [22, 107, 124]	136x136 to 1296x1296	milliseconds
Piezo [91]	576x576	milliseconds
Guided Wave [115]	16x16	nanoseconds
Wavelength [139]	100x100	nanoseconds
Tunable lasers + AWGR [13]	100x100	nanoseconds

Table 3.1: Comparison of different optical circuit switching technologies on port count and reconfiguration latency.

The fabric sends a global synchronization signal to announce the start of a new time slot (e.g., from a spine switch, with a leader selection and failover mechanism). This is straightforward in an electronic switch; optical circuit switches need a way to *broadcast* a time sync signal to all end nodes simultaneously. For example, we can generate the time signal electrically, then convert it to an optical signal. The signal is broadcast to the end points using a 1-to-N passive coupler at the switch to couple the time slot signal into the optical links connected to the end nodes.

The time slot signal is only used to announce a new time slot; it is independent of the local clock driving the sequential logic in the accelerators and NICs (at, say, 1GHz). In practice, for electrical and optical switches, the range of time slot durations $1\mu s < T < 10ms$ and therefore corresponds to $10^3 - 10^6$ clock cycles of the GPU.

Accelerators will be at different distances from the spine switch and will "hear" about the start of the new time slot at different times. If the variation in latencies is small compared to T, this does not matter. The most efficient systems will measure the round-trip time to every device so that its data arrives at the correct time to the spine switch. The system designer can decide whether or not to implement such a mechanism based on the switching efficiency, $\eta = \frac{b/c}{T} \leq \frac{b/c}{b/c+t_{max}+t_x}$, which is the fraction of time useful data is transferred.

GPUs	Tensor	Context	Pipeline	Data
8,192	8	1	16	64
16,384	8	1	16	128
16,384	8	16	16	8

Table 3.2: Three different configurations of GPUs used to train the 405B parameter Llama 3 model, using four types of parallelism. The entry in the table indicates the degree of each type of parallelism.

Example T for Llama 3. The largest Llama 3 [45] model has 405B parameters and is trained using tensor, context, pipeline and data parallelism, in the three configurations in Table 3.2. Data parallelism generally transfers the smallest units of data and dictates the slot time. The amount of data is determined by, (a) the number of parameters in each layer (because of compute-communication overlapping [87], the communications kick off after the backward pass of a bucket finishes, usually for a single layer for the largest models), (b) the degree of tensor parallelism (because it shards the parameters within a layer) and (c) the degree of data parallelism as ring-based collectives divide the transmitted tensor into chunks. If we denote the parameters per layer p_l, then $b = \frac{4p_l}{TP.DP}$, where TP is the degree of tensor parallelism (first column in the table) and DP is the degree of data parallelism. The factor of four is because the gradients are in `fp32` format. With $2.8B$ parameters per layer, and a link speed of 800Gb/s, the lowest $\frac{b}{c}$ is 109μs across the three configurations. If t_{max} is 10μs (2km links) and t_x =10ns, then we lose 8.4% throughput.

More examples. A circuit switch that connects accelerators over 200m links running at 400Gb/s, and b = 1Mbyte, should use a time slot $T \geq 22\mu s + t_x$. For an electronic switch or an optical switch built with fast tunable lasers ($t_x = 10ns$) the efficiency exceeds 95%. If we replace the switch with an optical MEMS switch with $t_x = 1ms$, we need $b >$ 150Mbytes for the efficiency to exceed 75%.

3.6 Failures and Stragglers

It takes weeks to train a large model, and hardware failures are inevitable. Periodic checkpoints help, but checkpointing the training state is expensive and takes time. If the operator checkpoints too often then training takes too long to complete; not often enough and training has to be frequently repeated.

Example. Meta reported 419 unexpected interruptions during a 54-day training run of Llama 3 with an MTBF (mean time between failure) of about three hours [45]. 58.7% of failures were GPUs (HBM3 issues, SRAM issues, and faulty GPUs), 8.4% were switches and cables, and 1.7% were NICs.

Before adopting this extreme approach (i.e. using an AI fabric built entirely from circuit switches), an operator would need to understand the consequences of hard failures (e.g. components failing, links breaking) and soft failures (e.g. straggler GPUs that respond slowly or inconsistently, links that are flapping on and off, or causing many bit errors).

Answering these questions fully is beyond the reach of our work, until we design a more complete system prototype, and so we leave the comprehensive analysis of failures for further work. However, there is some reason to think the MTBF of individual components will be about the same: The system will have the same number of GPUs, CPUs, memories, and NICs as before. In principle, a circuit switch is more reliable than a packet switch because it is simpler, particularly if it is electronic. However, given that the system component count is dominated by other components, this is unlikely to affect the overall system MTBF.

If the AI fabric fails, or if GPUs miss their time slot, the recovery process will be different for a circuit switch than for a packet switch. Generally, packet switches will be more forgiving of timing errors; on the other hand, it is easier to determine an error in a circuit switch when data is expected to arrive at a pre-determined time. A full system design needs to consider the consequences of different types of error, and map in appropriate standby components as needed.

In future work, we plan to study if and how the whole system can be stopped and started synchronously, by controlling the flow of synchronization messages. If possible, this might allow the system to be frozen, interrogated, and possibly healed without losing system state.

The switch fabric also needs to handle variations in GPU processing time - for example, when a GPU takes longer to finish its calculation than expected. A simple solution is to delay the next time slot until all data has arrived. Delays would reduce efficiency and increase the overall training time, and so the system controller will need to decide how often this can be allowed, and by how much, before intervention or replacement is required.

Failure tolerance. There have been several proposals to handle failures during training [40, 53, 121, 10, 133]. We argue that the failure tolerance of our design is no worse than packet switching, and any additional traffic during failures can be scheduled using the BvN decomposition.

Stragglers. Stragglers in training have been investigated in prior work [70, 131, 68, 130]. While it is easier to identify stragglers in our design (due to their non-adherence to the time slots) than the status quo, a definitive answer to the straggler tolerance of our design requires more study.

3.7 Handling unpredictable traffic

The programmer does not always know about a particular communication in advance. For example, control and management messages, error messages, interrupts and reloading data after a checkpoint. A big and important example is when the system uses a mixture-of-experts (MoE) model. The routing of data to experts is implemented using the *all-to-all* collective and its input-dependence makes it hard to predict or bound the patterns and volume of communication.

Mixture-of-experts splits the transformer's feedforward network (FFN) into multiple smaller FFNs (from 8-128 *experts*) and in *expert parallelism* the experts are placed on different accelerators. A routing layer computes the data (i.e., tokens) to send to each expert, which tells us the traffic

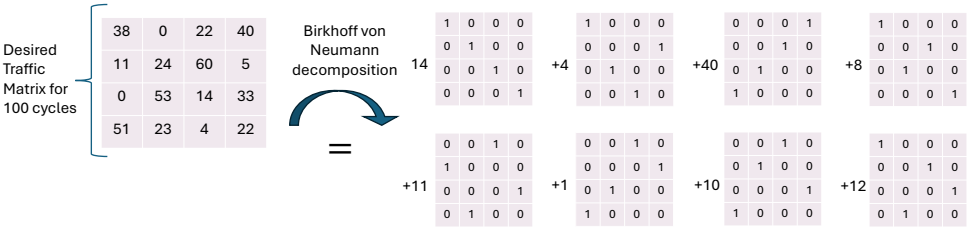


Figure 3.3: The Birkhoff-von Neumann decomposition of a traffic demand matrix into permutations. The weight corresponding to each permutation indicates how many times the permutation is held.

demand matrix between the GPUs.

Whenever the system needs to carry unpredictable traffic, we need to configure the crossbar switches on demand. We can not use packet switching, because the switches contain no buffers or forwarding logic. Instead, we can calculate new permutations on the fly to carry the traffic pent up in the network interfaces' VOQ.

Luckily, there is a clever way to turn a traffic matrix into a sequence of permutations called a Birkhoff-von Neumann (BvN) decomposition [18, 127]. Figure 4.1 shows a simple example. Decomposing an $N \times N$ matrix produces $O(N^2)$ unique permutations; each permutation is found using a bipartite matching algorithm. Each permutation is, in turn, subtracted from the traffic matrix to create a residual matrix. The permutation is held for consecutive time slots equal to the minimum entry along the permutation.

If we run a maximum cardinality matching algorithm at every step (e.g., the Hopcroft-Karp algorithm [48]), the overall time complexity of a BvN decomposition is $O(N^{4.5})$. We can instead use a much faster *maximal* (greedy) algorithm [117], which is close to optimal (maximum size). The times to compute the BvN decompositions on an Apple M1 Max CPU are reported in Table 3.3. Performance can be improved with specialized hardware, in particular using the Wrapped Wave Front Arbiter (WWFA) algorithm [117], which is known to be the most hardware-efficient maximal matching algorithm.

We invented a novel hardware algorithm (*BhaVaN*) that is the fastest known BvN decomposition. BhaVaN is implemented in Verilog and uses the WWFA and completes a BvN decomposition in less than 1μs for a 64-port switch on a 16nm ASIC process. The design takes up less than 1mm^2 and hence would consume less than 1% of an electronic circuit switch ASIC. If BhaVaN is used with an optical circuit switch, the algorithm can run on a modified NIC chip. BhaVaN will be described in more detail in Chapter 4.

Time slots. Unpredictable traffic can use a number of successive time slots dictated by the permutation weights in the BvN decomposition. For example, if the weight is two, then the permutation is held steady for two time slots.

	N=16	N=32	N=64	N=128	N=256
Maximum	0.5ms	3ms	25ms	291ms	4.45s
Maximal	0.2ms	0.8ms	4.6ms	52ms	0.65s

Table 3.3: Time to compute BvN decomposition in single-threaded software for $N \times N$ matrices, different N.

3.7.1 Performance comparison

We compare the completion time of MoE traffic for our circuit-switched approach against conventional packet switching. We do this by generating 1,000 different MoE traffic matrices and simulate how long the packet-switched network takes to transfer data, and how long the BvN decomposition takes to run for circuit switching, plus the transfer time.

Generating MoE traffic matrices. We generate sample MoE traffic matrices using the data measured by 1,000 successive iterations of training of a 16-expert GPT-MOE system [39]. The matrices are quite non-uniform, with 4 out of 16 experts receiving up to 80% of the tokens (Figure 3.4).

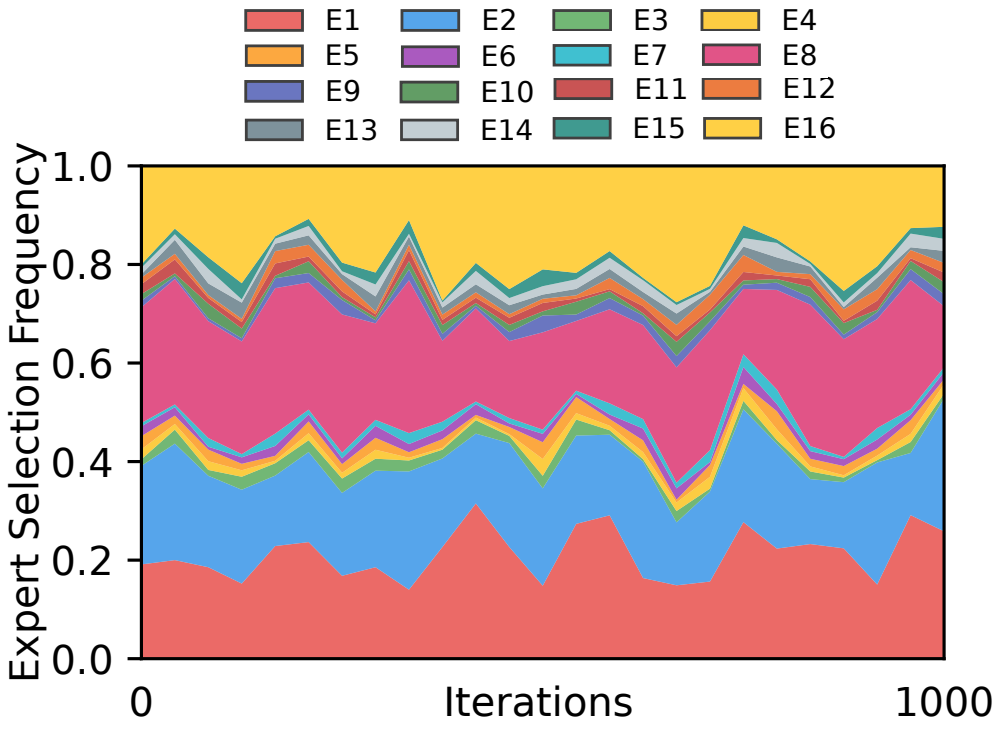


Figure 3.4: The fraction of tokens received by each expert across successive training iterations for a GPT-MoE model. E1 refers to expert 1. Figure reproduced from [39].

To generate a 16×16 traffic matrix, we use the fraction of tokens received by each expert and randomly spread them across the source GPUs from which the tokens arrive.[4] We repeat this for every iteration in the training data, to generate 1,000 traffic matrices.[5] We use a hidden size of 16,384 and a sequence length of 8,192 as in Llama 3 405B [45]. Figure 3.5 shows an example traffic matrix and the amount of data transferred (in megabytes).

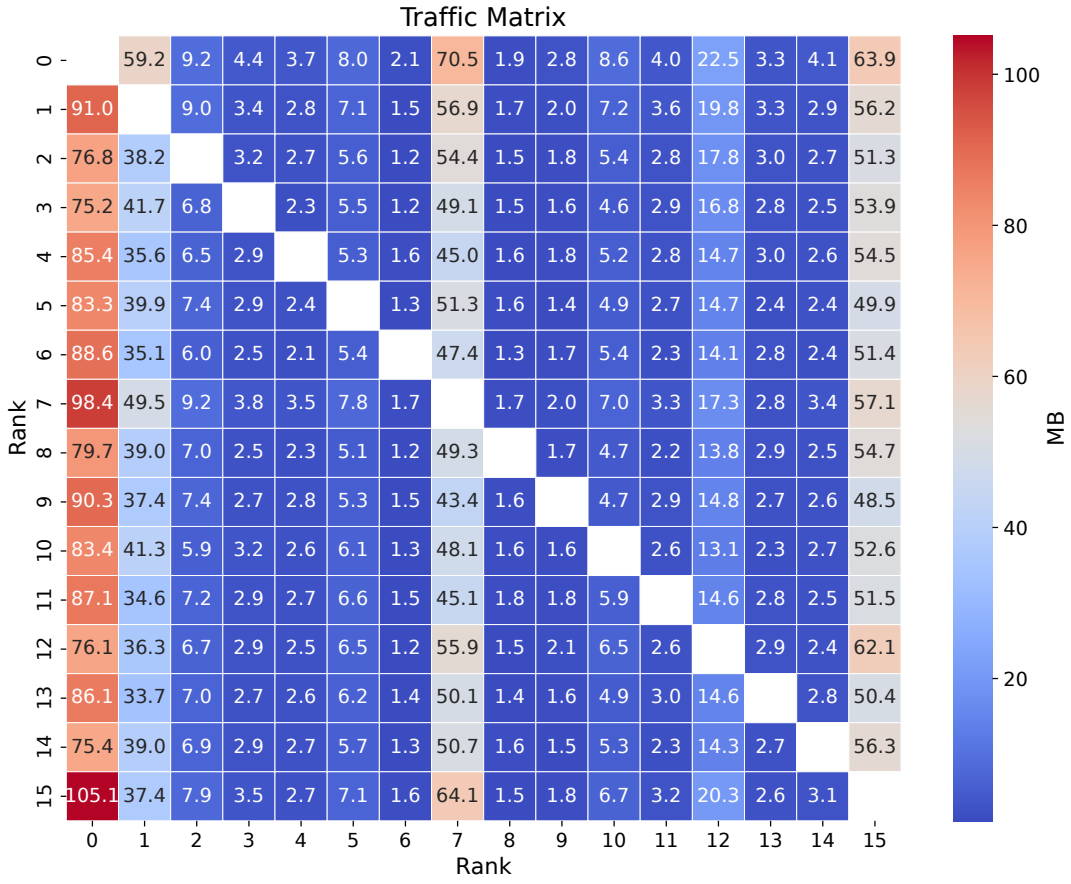


Figure 3.5: Example generated traffic matrix.

Topology and performance metric. The GPUs that host the experts are typically connected to the same switch. Our simulations assume a link speed of 800 Gb/s and an RTT of 1 μs. Our performance metric is the all-to-all completion time.

Simulations. The packet switch is simulated using ns-3 from the HPCC ns-3 repo [6], with DCQCN [146, 12] as the congestion control. The circuit switch completion time is calculated by

[4] If multiple experts are placed on a single GPU (e.g., 4 experts per GPU [99, 50]), the traffic matrix is smaller and easier to decompose compared to when all the experts are placed on different GPUs.

[5] While not strictly independent samples (they are from the same training run), we will assume they are, and we use them for 1,000 Monte Carlo simulation runs.

adding the permutation weights of the BvN decomposition and dividing by the link speed. For a fair comparison, we tune DCQCN parameters (K_{min}, K_{max}, P_{max}, R_{AI}, R_{HAI} and g) using a hyperparameter optimization framework [17, 51].

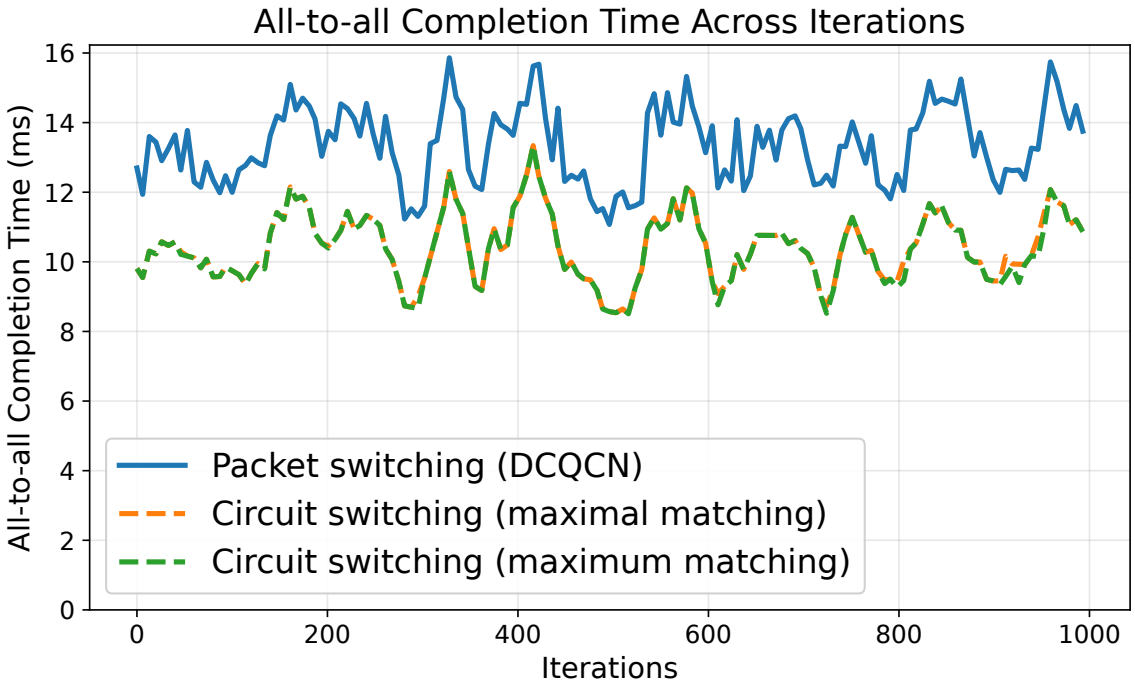


Figure 3.6: Comparison of the all-to-all completion times for 16 experts, for 1,000 iterations.

Figure 3.6 shows the time to transfer all data to 16 experts, over 1,000 runs, each with a different traffic matrix. The average completion time for packet switching is approximately 13ms, with clear peaks during times of congestion. The circuit-switched network finishes 22.04% faster on average (max 32% faster) because there are no packet buffers and data is transferred in an orderly fashion. Furthermore, once the relatively short flows to a given destination depart, the packet switching control loop is slow to grab the available bandwidth for the remaining large flows due to uplink contention at the senders.

Note that the time to decompose the traffic matrix ($<$ 0.5ms for 16 GPUs) is small compared to the transfer time, and very efficient in this case, with maximal or maximum matching. If MoE starts to use 256 or more GPUs in the future, we will need to implement it in hardware, as described above.

3.8 Discussion

This chapter explored the feasibility of building a complete training system using a circuit-switched fabric, addressing three key questions. First, we demonstrated that the majority of training traffic is predictable and can be deduced from the training code through our Genstack tool. Second, we showed that unpredictable traffic can be scheduled on the fly fast enough using a parallel algorithm (BhaVaN, described in Chapter 4) that schedules traffic in less than 1μs. Third, we argued that the design should handle failures and stragglers no worse than packet switching, though this requires further study. An important unresolved question is quantifying the power, cost, and area benefits compared to packet switching, which remains a valuable direction for future work as it requires prototyping and detailed hardware analysis.

Our proposal focuses on large single-job training runs, but it can be applied to multi-tenant training clusters by leveraging the insights in prior work [97, 145], where training jobs are interleaved so that their communications do not interfere.

Chapter 4

Fast BvN decomposition in hardware for circuit switches

4.1 Introduction

There is growing interest in using optical circuit switches in AI clusters, to reduce power, the number of electrical-to-optical and optical-to-electrical conversions, and to increase switching capacity. Google's Lightwave network [72] and TopoOpt [129] showed that using optical circuit switches in the upper tiers of the network hierarchy can reduce power and speedup training by eliminating congestion. As described in Chapter 3, Chronos explores taking this approach to the extreme, arguing that it may now be feasible to replace *every* packet switch in a modern AI cluster with simple and fast electronic or optical circuit switches. Regardless of whether AI clusters will replace *all* packet switches with circuit switched crossbars, it seems likely that optical circuit switches will be increasingly used deeper into the cluster [90, 57, 84].

Several authors have observed that the majority of training traffic is completely predictable: We know exactly when a GPU will send data, to whom and how much, up-front before training begins [97, 69, 98, 128]. For the most common modes of parallelism—data parallelism [100], tensor parallelism [111], and pipeline parallelism [49, 82]—Chapter 3 demonstrated how to analyze training source code using the Genstack [103] tool to generate a sequence of permutations that configure circuit switches to transfer the correct data during training.

Not all AI training traffic is predictable. AI clusters must also carry *unpredictable* traffic such as control and management traffic, checkpointing, and traffic for Mixture-of-Experts (MoE) parallelism [32]. Today, MoE communication can be up to 20% of training time [55]. If we are to use circuit switches, we need to find a way to quickly pick a sequence of permutations to deliver the newly arrived MoE traffic. Luckily, MoE traffic is relatively local, between 64-128 GPUs within a

cluster, allowing the decision to be made locally. Our approach is for GPU NICs to make requests to a local scheduler, which decides a sequence of permutations to deliver the traffic.

A natural approach is to use a crossbar scheduler, similar to those used in Internet routers (e.g. PIM [9], iSLIP [77] or WFA [117]). A NIC makes a request every time slot, which is the time taken to send the smallest data unit—about 1μs in the big AI training systems discussed in Chapter 3. Every time slot, a central scheduler chooses a permutation and sends back a grant to tell the NIC to send traffic in the next time slot.

But AI training machines are heavily pipelined and the delay between a request and grant can add significant overall latency to the training, particularly as the fraction of MoE traffic grows. Instead, we explore scheduling many consecutive time slots at the same time: The scheduler is given a matrix of all outstanding requests and then generates in one shot a sequence of, perhaps hundreds or thousands, of permutations to transfer all the traffic.

This is an example of a Birkhoff-von Neumann (BvN) decomposition algorithm [18, 127], a complex algorithm that takes a while to run. BvN has been used to schedule switches that change slowly and infrequently [24, 25, 93]. Unfortunately, the canonical BvN is about five orders of magnitude too slow for our needs.

The BvN decomposition algorithm proceeds in $O(N^2)$ rounds where a bipartite matching algorithm is run in each round leading to runtime complexity of $O(N^{4.5})$ for an $N \times N$ matrix. As we will see in Figure 4.2, existing BvN approaches, even with dedicated hardware, are orders of magnitude too slow.

This chapter addresses the question: *How fast can a Birkhoff-von Neumann decomposition algorithm run?*

Our solution starts with the Wrapped Wave Front Arbiter [117] as a key building block that uses several forms of parallelism simultaneously. The BhaVaN algorithm extends WWFA by adding bitwise parallelism in dedicated hardware. Our Verilog design shows that BhaVaN can decompose an $N \times N$ traffic matrix (with each value being an M-bit integer) into approximately $2^M * N$ unit-weight permutations in $2N-1$ clock cycles, *independent of* M.[1] BhaVaN can issue a new BvN decomposition (i.e., another set of about $2^M * N$ permutations) every N clock cycles using pipelining. We synthesized BhaVaN using a TSMC 16nm library, for which a 256×256 traffic matrix can be decomposed in 512ns.

For a 3nm process it will run even faster—we expect it to run in less than 300ns. The design is small: For a 256×256 traffic matrix with $M = 16$ (i.e., the largest value in the matrix can be $2^{16} - 1$), the estimated total area is approximately 16 mm^2 on a 16nm process and we project it to be 1 mm^2 on a 3nm process.

To put this into perspective, if we know in advance that the MoE cycle of computation generates up to (say) 1024 requests per NIC, then all 1024 time slots of traffic can be scheduled in less than

[1] Note that M is a factor in the area and power of the algorithm, not in the runtime.

1μs—less than one time slot.

The key contributions of this chapter are as follows. First, we present BhaVaN, a hardware algorithm for Birkhoff-von Neumann decomposition that operates at nanosecond timescales, enabling real-time traffic scheduling in circuit-switched networks. Second, we introduce bitwise WWFA parallelism, which allows all bit indices in the decomposition to be computed simultaneously rather than sequentially. Third, we provide a detailed hardware architecture and implementation analysis, showing that BhaVaN can be realized in less than 1mm^2 of silicon area while operating at gigahertz frequencies. Fourth, we compare the performance of BhaVaN with other matching algorithms (e.g., PIM [9], iSLIP [77], Hopcroft-Karp [48]) in the critical path of Birkhoff-von Neumann decomposition, and show that the cycle times (i.e., sum of weights of the permutation matrices) are comparable. Finally, we demonstrate that with BhaVaN, circuit switching becomes a viable alternative to packet switching for AI fabrics.

4.2 Birkhoff-von Neumann Decomposition

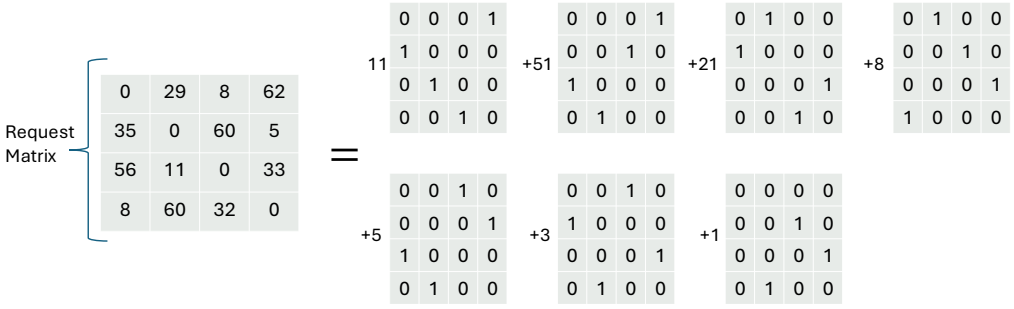


Figure 4.1: The Birkhoff-von Neumann decomposition of a request matrix into permutations.

Given an $N \times N$ matrix doubly stochastic matrix[2] D with the rows and columns adding to C, the Birkhoff-von Neumann (BvN) algorithm decomposes the matrix into a weighted sum of permutation matrices P_i such that $D = \sum \alpha_i P_i$ (in our case, all entries in D are non-negative integers and the weights α_i are positive integers). In each round i of the BvN decomposition, a maximum size bipartite matching is performed and a corresponding permutation matrix P_i is found. α_i is the smallest element of D in the permutation matrix P_i. The permutation is repeated α_i times in the decomposition, and $\alpha_i \times P_i$ is subtracted from D. The process is repeated until $D = 0$, which takes at most $O(N^2)$ iterations because each iteration takes at least one matrix value to 0.

An example decomposition of a 4×4 matrix is shown in Figure 4.1. The theorems by Birkhoff [18]

[2]A matrix is doubly stochastic if every row and column sums to the same value.

and Hall [46] prove that if a maximum cardinality matching algorithm is used (such as Hopcroft-Karp [48]) then D is decomposed into a minimum cycle of $\sum \alpha_i = C$ permutations.

Algorithm 1 Birkhoff–von Neumann Decomposition

Require: $D \in \mathbb{Z}_{\geq 0}^{N \times N}$
Ensure: Stack S of at most N^2 permutation matrices with weights
1: $S \leftarrow [\,]$ ▷ Empty stack of (P, α) pairs
2: **while** D has a nonzero entry **do**
3: $P \leftarrow \text{BIPARTITEMATCHING}(D)$ ▷ P is the permutation matrix from the bipartite matching
4: $\alpha \leftarrow \min\{D[i,j] \mid P[i,j] = 1\}$ ▷ Minimum value along the matching; $\alpha \in \mathbb{Z}_{>0}$
5: $D \leftarrow D - \alpha \cdot P$ ▷ Subtract α along matched edges
6: Push (P, α) onto S
7: **end while**
8: **return** S

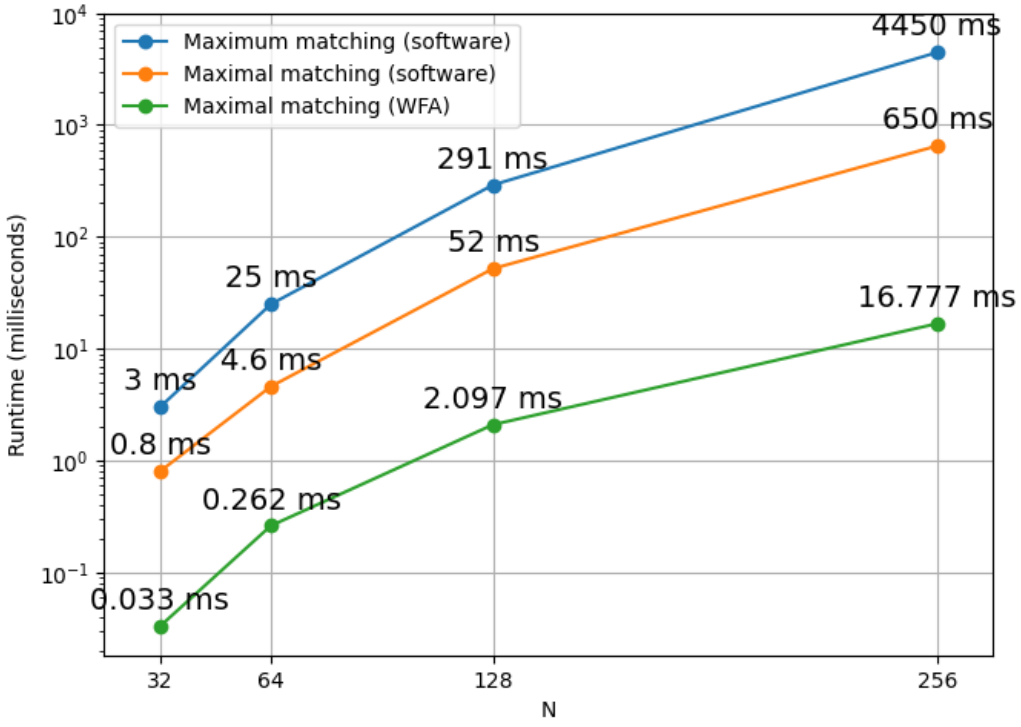


Figure 4.2: Running times of BvN decomposition for increasing N: (a) in software using maximum cardinality matching, (b) in software using maximal matching, and (c) in specialized hardware using WFA (Approach 2).

The pseudocode for the algorithm is given in Algorithm 1. The time complexity of the Birkhoff-von Neumann decomposition algorithm depends on the time complexity of the matching algorithm used. If Hopcroft-Karp matching [48] is used ($O(N^{2.5})$), then the overall time complexity is $O(N^{4.5})$. If

instead the Hungarian algorithm (Kuhn-Munkres, $O(N^3)$ variant) [65, 81] is used the overall time complexity is $O(N^5)$.

Software implementations are far too slow for real-time applications such as MoE models which require a new decomposition to happen on the order of microseconds. We can see from Figure 4.2 that it takes 100's of *milliseconds* to decompose an 128×128 matrix on a modern Apple M1 Max CPU. A faster matching algorithm [43] would help reduce the overall time complexity to $O(N^3 logN)$. However, even these algorithms are far too slow to be able to perform decompositions in microsecond timeframes.

4.3 Speeding up the Birkhoff-von Neumann decomposition

The challenge is that the decomposition algorithm in Algorithm 1 has a sequential dependency: it must find the minimum value (α) before starting the next iteration. This limits parallelism.

We overcome the sequential dependency in BvN by introducing several forms of parallelism. First, we replace the maximum cardinality matching algorithm (Hopcroft-Karp or Hungarian) with a much simpler, greedy, online *maximal* matching algorithm. Maximal matching is much faster and simpler, and can be implemented by iterative algorithms (e.g., PIM [9], iSLIP [77], etc.) or feedforward wave front arbiters (e.g., WFA and WWFA [117]).

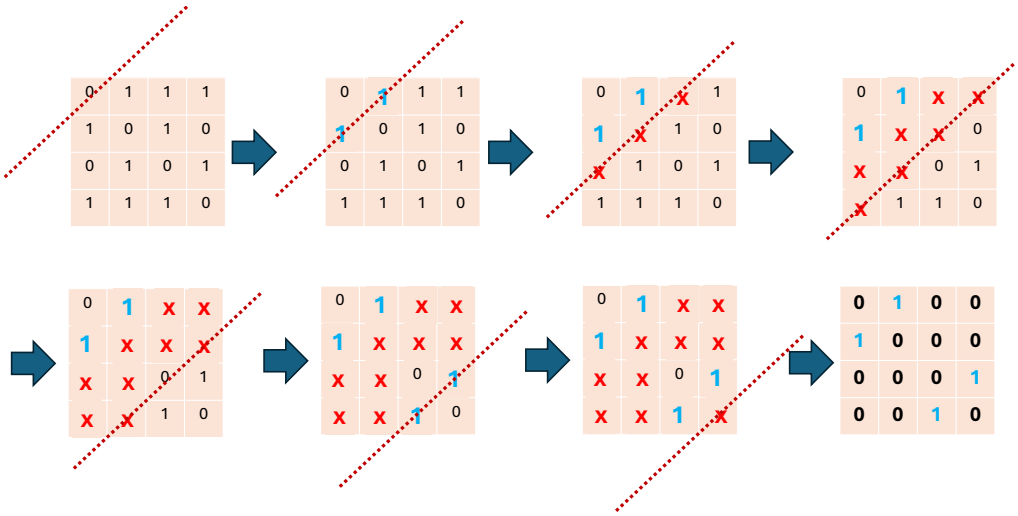


Figure 4.3: A worked example showing the operation of the wave front arbiter (WFA) algorithm. Elements marked in blue are grants and elements marked in red are conflicts with existing grants. The final permutation produced by WFA is shown in the bottom right.

Our approach uses WFA as a key building block. WFA is the fastest known hardware algorithm to find a maximal bipartite matching. It operates as a diagonal wave front propagating through the matrix from top left to bottom right. Figure 4.3 shows an example of WFA. WFA lends itself to

fast hardware because it exploits two forms of parallelism: First, all entries in a diagonal can be processed at the same time because they do not conflict with each other, and second, because it is a feedforward algorithm it can be easily pipelined. In pipelined WFA, each diagonal is occupied by a different wave front with $2N-1$ wave fronts active at the same time. WFA can calculate a single maximal matching in $2N-1$ clock cycles, and then a new match can be issued *every clock cycle* thereafter [117]. In a packet switch, care has to be taken because the WFA algorithm naturally favors the top left hand corner and can systematically starve some input/output pairs indefinitely. In practice, this requires complex permutations of the input/output pairs to give long-term fairness. But, for Birkhoff-von Neumann decomposition, we do not need to worry about this because the algorithm runs to completion and all sequences of permutations are equivalent. In our design, we use the Wrapped Wave Front Arbiter (WWFA [117]) because it completes in half the time of WFA, but is a little harder to illustrate.

4.3.1 Adapting WWFA for BvN decomposition

While WWFA is the fastest known hardware algorithm to find a maximal bipartite matching, we must adapt it for BvN decomposition. This is because when WWFA finds a permutation, we have to find the minimum value in D corresponding to the permutation; this is tricky because WWFA is a feedforward algorithm. The wavefront cannot subtract the correct values from the request matrix until the wavefront reaches the end of the matrix.

There are two ways in which we can use WWFA in a BvN decomposition.

Approach 1: Subtract the value '1' each time

The first approach is to subtract the value 1, instead of the minimum value in D corresponding to the match. In this approach, each permutation has unit weight, and hence we require C permutations, with the run time being proportional to C as well. The cycle of permutations must be post processed to form a stack with larger weights. The tradeoff is therefore maximally pipelining WWFA (good), but a longer run time (bad).

Approach 2: Flushing the pipeline to find the exact minimum value

The second approach sacrifices pipelining to find only one permutation at a time. Only one wavefront is in the array at a time, so that we can explicitly calculate the minimum value (and hence the weight) before WWFA finds the next permutation. The time complexity of this approach is $O(N^3)$ where each WWFA run takes up $O(N)$ time, and there can be $O(N^2)$ permutations in the worst case. This approach provides a bounded runtime for matrices with large values. However, even with a bounded runtime, this approach could take 10's of milliseconds even in specialized hardware for $N = 256$ as shown in Figure 4.2.

4.3.2 BhaVaN's novelty: Bitwise-parallel WWFA

BhaVaN uses bit-wise parallelism to completely sidestep the problem described above. The idea is quite simple: If the values in D are M-bit integers, we replicate WWFA M times (i.e. M separate hardware instances, one per bit). The ith WWFA instance only operates on bit position i of the request matrix values. There is no dependency between the WWFA instances and each of them can operate in parallel. This allows us to subtract a fixed minimum value of 2^i immediately for the ith instance thereby enabling pipelining both within a matrix and across matrices.

```
// For each bit position b (in parallel)
for (b = 0; b < M; b++) {
    // Extract binary request matrix
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            request[b][i][j] = (D[i][j] >> b) & 1;
        }
    }
    // Run WWFA on bit position b
    while (any requests remain at bit b) {
        permutation = WWFA(request[b]);
        output_permutation(permutation, weight=2^b);

        // Clear granted requests
        for each (i,j) in permutation {
            request[b][i][j] = 0;
        }
    }
}
```

Listing 4.1: BhaVaN Core Algorithm (Conceptual)

The operation of BhaVaN is shown in Figure 4.4. Bitwise parallelism enables BhaVaN to achieve a bounded runtime of $2N - 1$ clock cycles for an $N \times N$ matrix. The bounded runtime is because a binary request matrix of size $N \times N$ can be decomposed into at most N permutations (leading to at most $M * N$ unique permutations across all the bit planes, with permutations coming from the ith bit plane having a weight of 2^i thereby resulting in about $2^M * N$ unit-weight permutations). Listing 4.1 shows pseudocode for BhaVaN.

In Section 4.5 our Verilog synthesis suggests that BhaVaN can sustain a clock speed of 1 GHz, resulting in a runtime of $2N - 1$ ns. Compared to the pipelined Approach 1 above, BhaVaN provides a runtime speedup of V (where V is the largest value in the matrix) for a $log_2(V)$ increase in ASIC area. On the other hand, compared to the non-pipelined Approach 2 above, BhaVaN is faster by a

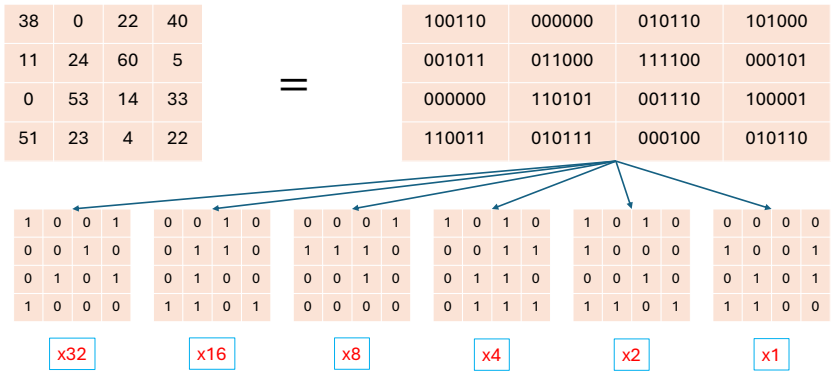


Figure 4.4: High-level working of BhaVaN where $M = log_2(V)$ (V is the maximum value in the matrix D) instances of WWFA operating in parallel with the ith instance operating on bit plane i, with the permutations coming out of bit plane i having weight 2^i (shown at the bottom). $M = 6$ in the example shown.

factor of O(N^2). To our knowledge, BhaVaN is the fastest ever BvN decomposition algorithm by several orders of magnitude.

4.4 Performance of maximum size vs maximal algorithms

We can compare different BvN decomposition algorithms by the number of time slots needed to transfer all the traffic (i.e., the cycle length) in the request matrix. We know from the results of Birkhoff [18] and Hall [46] that maximum cardinality matching yields the fewest time slots. How do maximal matching and BhaVaN compare in terms of the number of time slots? If we need extra time slots, we have to run the system faster to make up for it.

It is known that a maximal bipartite match is at least half the size of a maximum size match [126]. Intuitively, this suggests that BvN needs at most twice as many time slots if it use maximal matches instead of maximum cardinality matches. This is indeed the case, and is proved in Section 4.4.1.

In practice, for traffic matrices generated uniformly at random (as they are likely to be in Mixture-of-Experts models), we found that the number of time slots needed by maximal matching (ie. WWFA) is much shorter than twice the optimum. Figure 4.5 shows the results for 1000 such matrices. Note that these results are for a software-based BvN decomposition that uses an $O(N^2)$ maximal matching algorithm in the critical path.

Interestingly, although BhaVaN runs the maximal matching WWFA on each bit plane, we found that the permutations it produces may not be maximal in pathological cases. Hence, the 2x bound does not apply to BhaVaN. One such example is shown in Figure 4.6, where we find that the cycle length of the decomposition produced by BhaVaN is 2.25× the minimum possible cycle length found

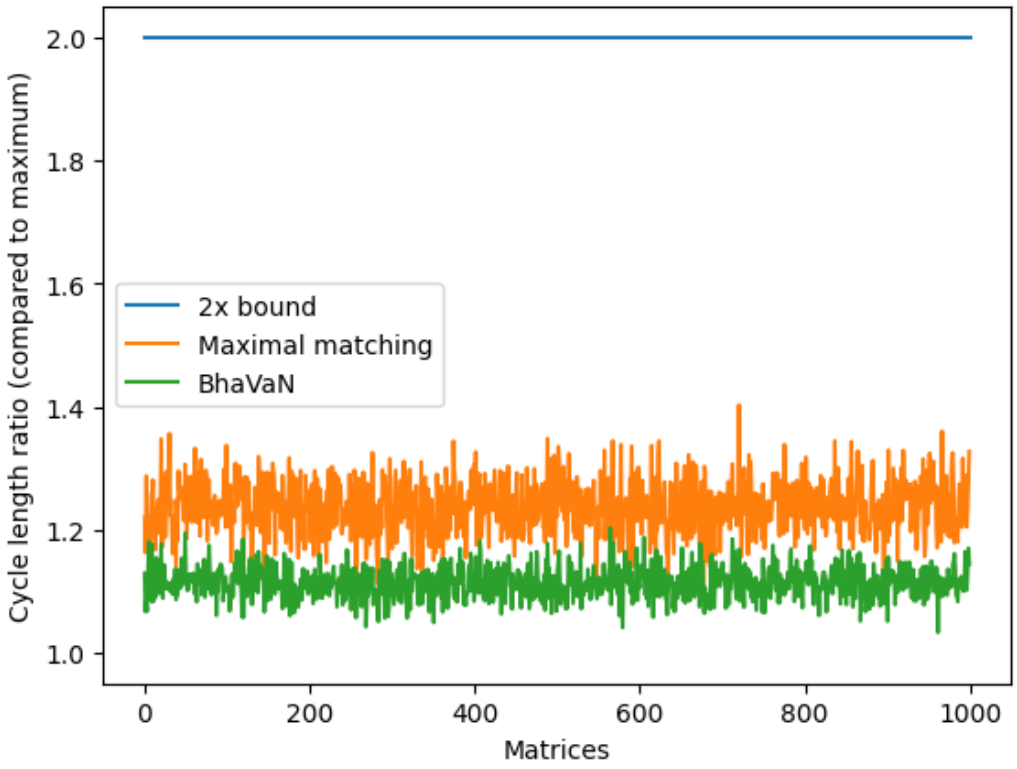


Figure 4.5: The number of time slots for maximal matching-based BvN and BhaVaN on 1000 randomly generated matrices, reported as a ratio compared to the minimum possible.

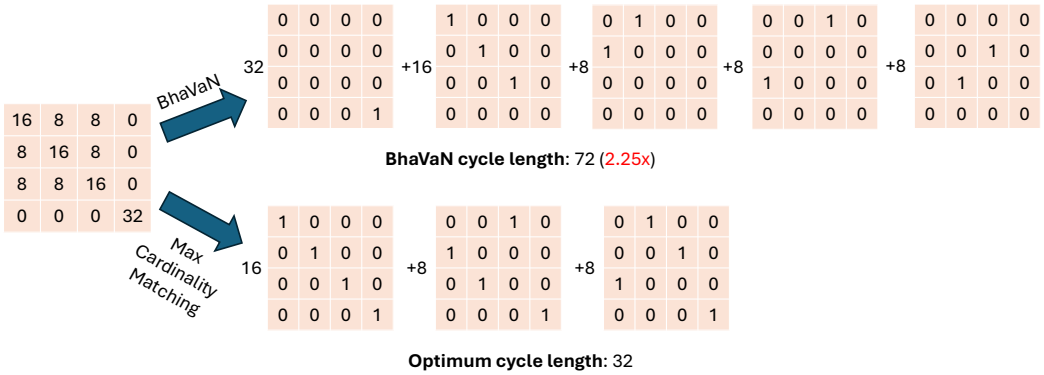


Figure 4.6: A pathological case where BhaVaN performs worse than the 2× bound. In addition to BhaVaN, the maximum cardinality matching based decomposition is also shown.

by maximum cardinality matching. In general, if the values of the matrix are such that a bit plane is only sparsely occupied, then we found BhaVaN is unable to find maximal matches and cycle lengths for those matrices can be larger than the 2× bound. More study is needed to understand the exact matrix structures for which BhaVaN produces matchings that are not maximal and performs worse than the 2× bound.

In summary, BhaVaN does not meet the 2x bound as shown in the counterexample above, but in practice we find it is actually *better* than maximal matching 99.8% of the time as shown in Figure 4.5.

4.4.1 Proof of BvN decomposition performance of maximal vs maximum size matching

Below is the theorem and proof of the fact that using maximal matchings at every step will produce at most twice the number of permutations (unit-weight) as using maximum size matchings at every step. A sharper bound is proved below.

Theorem 2 (Cycle length bound for maximal matchings in BvN decomposition)**.** *Let $A \in \mathbb{Z}_{\geq 0}^{n \times n}$ be a nonnegative integer matrix, and let*

$$\Delta \;=\; \max\left\{ \max_{i \in [n]} \sum_{j=1}^{n} A_{ij},\;\; \max_{j \in [n]} \sum_{i=1}^{n} A_{ij} \right\}$$

be the maximum row or column sum of A. We define the cycle length *of a Birkhoff–von Neumann style decomposition of A as*

$$L \;=\; \sum_{t=1}^{T} w_t,$$

where in each round t we select a matching M_t in the current residual bipartite multigraph, set

$$w_t \;=\; \min_{(i,j) \in M_t} \big(\textit{residual weight on edge } (i,j)\big),$$

and subtract w_t along all edges of M_t. The process terminates once the matrix is reduced to the all–zero matrix.

Let $L_{\max}$ denote the cycle length when each round M_t is chosen to be a maximum cardinality matching *in the residual graph, and let L_{mxl} denote the cycle length when each round M_t is only required to be a* maximal matching.

Then the following inequality always holds:

$$\frac{L_{\mathrm{mxl}}}{L_{\max}} \;\leq\; 2 - \frac{1}{\Delta}.$$

Proof. Given an $N \times N$ non-negative integer matrix $A = (a_{ij})$, build the bipartite multigraph

$G_0 = (U, V, E_0)$ where $|U| = |V| = N$ and a_{ij} parallel edges between $i \in U$ and $j \in V$. Δ is the maximum degree of a vertex in G_0 (in other words, the maximum row or column sum of A).

A BvN round that subtracts 1 along a matching corresponds to removing the edges of one *color class* in an edge-coloring and the *cycle length* equals the number of such rounds (or colors) [1]. That is, we start from G_0 and for rounds $t = 1, 2, \cdots$, we take any maximal matching M_t in the current residual graph G_{t-1} and delete the matched unit edges to obtain G_t, and continue until no edges remain.

Notation. Fix an initial unit edge $e = uv \in E_0$ where $u \in U$ and $v \in V$. Going by the following basic definitions:

- $\delta(u)$: set of edges incident on u in G_0.
- $\delta(v)$: set of edges incident on v in G_0.
- $d(u)$: number of edges incident on u in G_0.
- $d(v)$: number of edges incident on v in G_0.
- $\mu(uv)$: the number of *parallel unit edges* between u and v in G_0.

Then let

$$S_e \triangleq (\delta(u) \cup \delta(v)) \setminus \{e\}$$

Then $|S_e| = d(u) + d(v) - \mu(uv) - 1$.

Lemma 1 (Edge-local deadline for each edge)**.** *Under the maximal-match-every-round process, $e = uv \in E_0$ is deleted no later than*

$$\tau(e) = d(u) + d(v) - \mu(uv)$$

that is, by round $\tau(e)$ counting from round 1.

Proof. Suppose e survives k rounds. For any $r \in \{1, \ldots, k\}$, e is present in G_{r-1}. If M_r contained no edge incident to u or v, then we could add e to M_r and still have a matching, which contradicts the maximality of M_r. Hence, each round $r \leq k$ contains some edge $f_r \in M_r$ with $f_r \in S_e$.

Matches in distinct rounts remove distinct unit edges, so the f_r's are all distinct elements of S_e. Therefore,

$$k \leq |S_e| = d(u) + d(v) - \mu(uv) - 1$$

In other words, if e has not been removed until the round counter counts up to $|S_e|$, it will be in the next round, that is, round $|S_e| + 1$ or $\tau(e)$. If it is not removed by the next round, then u and v are free and maximality will be violated. □

Corollary 2. *Let L_{mxl} be the total number of rounds (in other words, the cycle length) under the maximal-match-every-round process. The last edge is removed by the round*

$$L_{\text{mxl}} \leq max_{e=uv \in E_0}(d(u) + d(v) - \mu(uv))$$

Since for every edge $e = uv$, $d(u) \leq \Delta$, $d(v) \leq \Delta$, and $\mu(uv) \geq 1$,

$$L_{\text{mxl}} \leq 2\Delta - 1$$

On the other hand any scheduled requires at least Δ rounds because a vertex of degree Δ can be matched at most once per round. More specifically, the cycle length of maximum-size-match-every-round $L_{\text{max}} = \Delta$ from Birkhoff's theorem [18] which uses Hall's marriage theorem [46]. So,

$$\frac{L_{\text{mxl}}}{L_{\text{max}}} \leq \frac{2\Delta - 1}{\Delta} = 2 - \frac{1}{\Delta}$$

□

4.5 BhaVaN in hardware

BhaVaN is designed to run very fast in dedicated hardware. We created two versions: Python (for functional verification and study) and Verilog (to verify speed and area). The Python code takes a randomly generated matrix as input and generates the sequence of permutations following the BhaVaN algorithm.

4.5.1 Verilog implementation

Our hardware design consists of 250 lines of Verilog and 300 lines of C++, and uses Verilator [114] to perform cycle accurate simulations. The input-output pairs (1000's of them, with several corner case matrices like all zeros, all ones, identity matrices, etc.) generated by the software implementation are passed through the simulator to verify the correctness and timing (i.e., not only are the outputs correct but also that they were produced at the correct clock cycles). The Verilog code is parameterized by N and M (the number of bit planes). The key features of our Verilog implementation are:

- *Modular building blocks*: There are M identical copies of the WWFA module which operate in lockstep. Each copy produces its first permutation N clock cycles into the decomposition and subsequent permutations in successive clock cycles with the last (Nth) permutation being produced in clock cycle $2N - 1$.

- *Unit decrements*: We use unit decrements in each of the M WWFA blocks. An accumulation step at the end sets the weight of permutations produced by block i to 2^i.
- *Pipelining*: In addition to WWFA pipelining within a single input matrix, we also pipeline across request matrices to process a new input matrix every N clock cycles despite a decomposition needing $2N-1$ clock cycles to complete. The insight that enables this is that we can load in the jth wrapped diagonal of the next request matrix once all the N WWFA wave fronts have passed through that diagonal for the current request matrix. For instance, N clock cycles into the current decomposition, all the N wave fronts have passed through and processed the first wrapped diagonal, and therefore this diagonal can be loaded in from the next request matrix.

4.5.2 TSMC 16nm synthesis

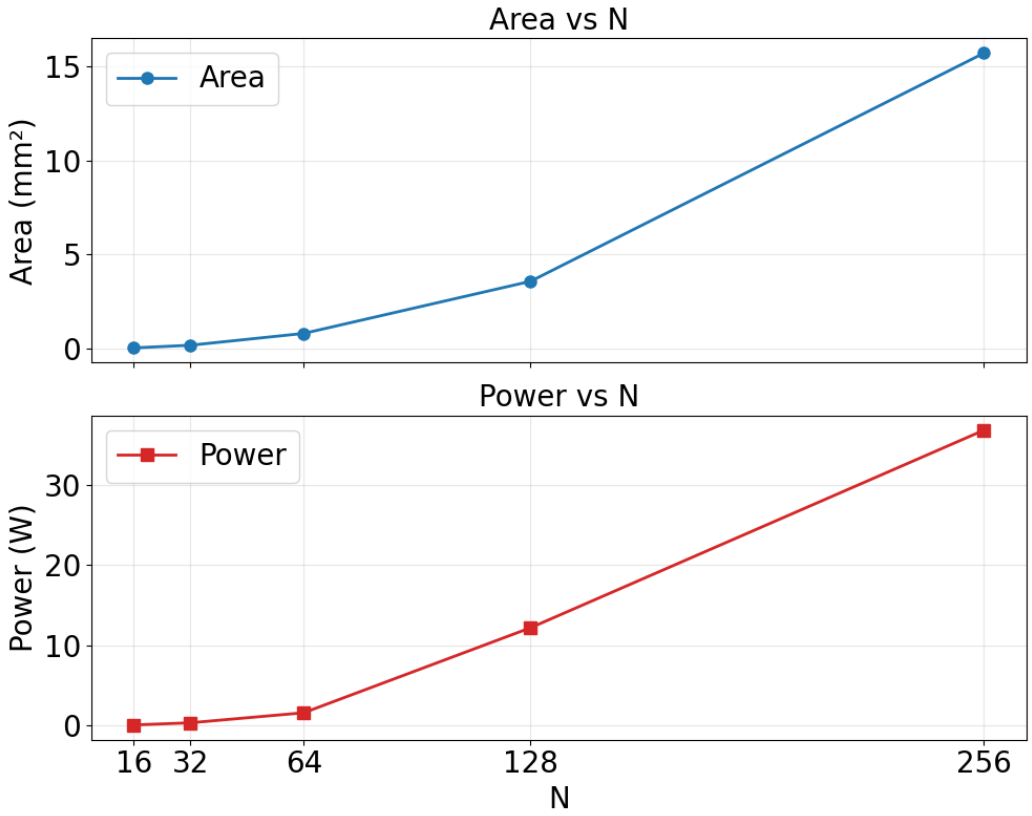


Figure 4.7: The area and power consumed by BhaVaN for increasing N using a TSMC 16nm process for M=16.

We synthesized the Verilog implementation using a TSMC 16nm library to understand the power, area and timing characteristics of the implementation. The area and power consumed by BhaVaN

for increasing N (and $M = 16$) is shown in Figure 4.7. We see that for $N = 256$, BhaVaN can be realized in about 16 mm^2 of area and consumes 36W of power. If we use a 3nm process, we project using publicly available numbers that the area consumed will be about 1 mm^2 and the power would be about 7W, which is small compared to state-of-the-art packet switching chips that are close to reticle-sized at 800mm^2 and consume hundreds of Watts.

4.6 Discussion and Summary

This chapter presented BhaVaN, a hardware algorithm for Birkhoff-von Neumann decomposition that operates at nanosecond timescales. The speed of BhaVaN opens up new possibilities to use BvN decomposition in real-time systems that were previously considered infeasible. Consider a modern high-speed router with 128×400Gb/s links: at minimum packet size (128B), each packet occupies a link for only 2.5ns. If we accumulate packets for 200ns (80 packet times), we obtain a request matrix with $M = 80$ (since values can reach up to 80). With $\lceil \log_2(80) \rceil = 7$ WWFA instances operating in parallel, BhaVaN can compute the complete BvN decomposition in just $2 \times 128 - 1 = 255$ clock cycles. At 1 GHz, this takes 255ns—comfortably within our 200ns accumulation window when pipelined. This demonstrates that BhaVaN can keep pace with line-rate traffic even at hundreds of gigabits per second, making it practical to use BvN decomposition in the critical path of packet processing for the first time.

The implications for AI training are particularly significant. MoE traffic, which can consume 20% of training iteration time, might otherwise force AI clusters to use higher power packet switches for unpredictable workloads. BhaVaN can schedule a 128-port fabric in under 256ns with 1mm^2 of silicon on a 3nm process. This means circuit switches can now handle both pre-scheduled traffic (via Genstack) and dynamic MoE bursts, reducing the power consumption of the AI fabric. BhaVaN makes circuit switching viable for a broad class of AI training systems.

Together with the work presented in Chapter 3, this chapter demonstrates that leveraging application knowledge—in this case, knowledge of traffic matrices and the structure of the Birkhoff-von Neumann decomposition—enables practical hardware solutions for real-time network scheduling at unprecedented speeds.

Chapter 5

Conclusions

This thesis demonstrates how we can improve networks by taking advantage of application-specific knowledge. We showed three new systems—Hydra, Chronos, and BhaVaN—that exploit knowledge of network properties, communication patterns, and traffic structure yielding orders-of-magnitude improvements in correctness, efficiency, and performance.

5.1 Dissertation Takeaways

The central contribution of this work is methodological: specialization, when grounded in stable application characteristics, can outperform general-purpose solutions. This insight manifests differently across our three systems, yet follows a common pattern: we identify what is predictable or specifiable in the application domain, and build a system to exploit that structure.

Hydra demonstrates that runtime verification at line rate is practical. By checking every packet in the data plane against a formal specification, it catches bugs that static analysis tools miss—implementation errors, hardware faults, and transient violations during updates. We demonstrated that it works by deploying in the Aether system and detecting real bugs in less than a microsecond.

Chronos demonstrates that circuit switching is worth considering for specialized workloads. For AI training, where 80% of traffic follows predictable patterns determined before execution begins, pre-scheduled crossbar switches eliminate 70% of switch logic while reducing completion times by 22%. The Genstack tool shows that we can extract schedules from training code. BvN decomposition can handle dynamic MoE traffic. The key insight is recognizing when workload predictability justifies architectural specialization.

BhaVaN demonstrates that algorithmic structure enables dramatic hardware speedup. Bitwise parallelism reduces BvN decomposition from $O(N^{4.5})$ software complexity requiring hundreds of milliseconds to $2N-1$ hardware cycles requiring hundreds of nanoseconds—a million-fold improvement. Occupying less than 1 mm^2 on 3nm processes, BhaVaN makes real-time circuit switching viable for

arbitrary dynamic traffic, completing the vision of circuit-switched AI fabrics.

The broader lesson: general-purpose networking, optimized for unpredictable heterogeneous workloads, leaves performance on the table for specialized applications. When application characteristics are stable and exploitable, purpose-built solutions can be simpler, faster, and more reliable.

5.2 Future Directions

Several promising directions could extend this work. For runtime verification, targeting diverse programmable substrates (eBPF, DPDK, SmartNICs) would broaden applicability, while hybrid static-runtime approaches might reduce overhead by statically proving properties when possible. It would be interesting to explore how the network could learn verification policies by examining traffic traces.

For circuit-switched training fabrics, we will only know the true benefits to cost and power by prototyping a real system. This would also enable us to study failures and reliability. Extending to multi-tenant clusters via temporal multiplexing and co-designing networks with training frameworks could improve utilization and performance. Applying these ideas to other domains—graph neural networks, distributed databases, recommendation systems—would test the generality of pre-scheduled circuits.

The impact of hardware-accelerated scheduling algorithms can be broadened by extending BhaVaN to multicast permutations, exploring alternative matching algorithms balancing approximation quality with hardware efficiency, and identifying other network optimization problems amenable to specialized hardware. It would also be worth investigating the integration with silicon photonics and co-packaged optics.

More broadly, developing principled methodologies for network specialization remains open. When should we build domain-specific network architectures versus using general-purpose solutions? How do we systematically derive network designs from application requirements? What abstractions enable specialization without sacrificing flexibility? The increasing diversity of network applications—from microsecond-latency trading to multi-day LLM training—suggests that specialization will become increasingly valuable, but the research community lacks systematic approaches to guide these design decisions.

5.3 Concluding Remarks

When this research began, it was unclear whether checking every packet at line rate was practical, whether circuit switches could handle diverse training traffic, or whether BvN decomposition could run in nanoseconds. This thesis showed all three are feasible.

The methodological contribution transcends the specific systems. Rather than accepting general-purpose solutions as inevitable, we can ask: what do we know about the application domain, and how can we exploit that knowledge? For network verification, we know desired properties. For AI training, we know communication patterns. For traffic scheduling, we know matrix structure. Systematically leveraging this knowledge yields qualitatively better solutions.

Modern networks face a tension: complexity is required for functionality, but complexity breeds bugs. This thesis presents two complementary approaches. Hydra manages complexity through verification—build complex networks as needed, but verify behavior continuously. Chronos manages complexity through simplification—for predictable workloads, use simpler architectures matching the application's structure.

The future of networking likely involves increased specialization. As diverse applications demand different performance characteristics—AI training requiring deterministic low latency, web services requiring elastic scaling, real-time video requiring bounded jitter—one-size-fits-all solutions become suboptimal. The challenge is developing principled approaches to specialization: identifying stable characteristics, deriving designs systematically, and validating sufficient benefits.

This thesis represents one step in that direction, demonstrating that runtime verification, prescheduled circuits, and hardware-accelerated scheduling are practical when we leverage application knowledge. The principle—specialize when characteristics permit, verify behavior not models, and optimize for actual workloads—provides a foundation for building the high-performance networks of tomorrow.

Sometimes, the extreme approach, carefully executed, becomes the practical one.

Bibliography

[1] G. Aggarwal, R. Motwani, D. Shah, and An Zhu. Switch scheduling via randomized edge coloring. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 502–512, 2003.

[2] Deepspeed AI. DeepSpeed repository, Fetched May 6th, 2025. `https://github.com/deepspeedai/DeepSpeed`.

[3] Lightning AI. PyTorch Lightning repository, Fetched May 6th, 2025. `https://github.com/Lightning-AI/pytorch-lightning`.

[4] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 19, 2010.

[5] Kinan Dak Albab, Jonathan DiLorenzo, Stefan Heule, Ali Kheradmand, Steffen Smolka, Konstantin Weitz, Muhammad Timarzi, Jiaqi Gao, and Minlan Yu. SwitchV: automated SDN switch validation with P4 models. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, pages 365–379, 2022.

[6] Alibaba. High-Precision-Congestion-Control repository, Fetched May 6th, 2025. `https://github.com/alibaba-edu/High-Precision-Congestion-Control`.

[7] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, Francis Matus, Rong Pan, Navindra Yadav, and George Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 503–514, 2014.

[8] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 113–126, 2014.

[9] Thomas E. Anderson, Susan S. Owicki, James B. Saxe, and Charles P. Thacker. High-speed switch scheduling for local-area networks. *ACM Trans. Comput. Syst.*, 11(4):319–352, November 1993.

[10] Daiyaan Arfeen, Dheevatsa Mudigere, Ankit More, Bhargava Gopireddy, Ahmet Inci, and Gregory R. Ganger. Nonuniform-tensor-parallelism: Mitigating gpu failure impact for scaled-up llm training, 2025.

[11] Manikandan Arumugam, Deepak Bansal, Navdeep Bhatia, James Boerner, Simon Capper, Changhoon Kim, Sarah McClure, Neeraj Motwani, Ranga Narasimhan, Urvish Panchal, Tommaso Pimpo, Ariff Premji, Pranjal Shrivastava, and Rishabh Tewari. Bluebird: High-performance SDN for bare-metal cloud services. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 355–370, 2022.

[12] Wei Bai, Shanim Sainul Abdeen, Ankit Agrawal, Krishan Kumar Attre, Paramvir Bahl, Ameya Bhagat, Gowri Bhaskara, Tanya Brokhman, Lei Cao, Ahmad Cheema, Rebecca Chow, Jeff Cohen, Mahmoud Elhaddad, Vivek Ette, Igal Figlin, Daniel Firestone, Mathew George, Ilya German, Lakhmeet Ghai, Eric Green, Albert Greenberg, Manish Gupta, Randy Haagens, Matthew Hendel, Ridwan Howlader, Neetha John, Julia Johnstone, Tom Jolly, Greg Kramer, David Kruse, Ankit Kumar, Erica Lan, Ivan Lee, Avi Levy, Marina Lipshteyn, Xin Liu, Chen Liu, Guohan Lu, Yuemin Lu, Xiakun Lu, Vadim Makhervaks, Ulad Malashanka, David A. Maltz, Ilias Marinos, Rohan Mehta, Sharda Murthi, Anup Namdhari, Aaron Ogus, Jitendra Padhye, Madhav Pandya, Douglas Phillips, Adrian Power, Suraj Puri, Shachar Raindel, Jordan Rhee, Anthony Russo, Maneesh Sah, Ali Sheriff, Chris Sparacino, Ashutosh Srivastava, Weixiang Sun, Nick Swanson, Fuhou Tian, Lukasz Tomczyk, Vamsi Vadlamuri, Alec Wolman, Ying Xie, Joyce Yom, Lihua Yuan, Yanzhao Zhang, and Brian Zill. Empowering azure storage with RDMA. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 49–67, Boston, MA, April 2023. USENIX Association.

[13] Hitesh Ballani, Paolo Costa, Raphael Behrendt, Daniel Cletheroe, Istvan Haller, Krzysztof Jozwik, Fotini Karinou, Sophie Lange, Kai Shi, Benn Thomsen, and Hugh Williams. Sirius: A flat datacenter network with nanosecond optical switching. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '20, page 782–797, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 44–57, 2004.

[15] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 155–168, 2017.

[16] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. Abstract interpretation of distributed network control planes. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2019.

[17] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. *Advances in neural information processing systems*, 24, 2011.

[18] Garrett Birkhoff. Tres observaciones sobre el algebra lineal. *Univ. Nac. Tucuman, Ser. A*, 5:147–154, 1946.

[19] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in Microsoft Azure. In *International Conference on Distributed Computing and Internet Technology (ICDCIT)*, pages 21–32, 2015.

[20] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[21] Zixian Cai, Zhengyang Liu, Saeed Maleki, Madanlal Musuvathi, Todd Mytkowicz, Jacob Nelson, and Olli Saarikivi. Synthesizing optimal collective algorithms. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 62–75, New York, NY, USA, 2021. Association for Computing Machinery.

[22] Calient. Calient.AI, Fetched November 4th, 2025. `https://www.calient.net/`.

[23] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, and Jennifer Rexford. A NICE way to test OpenFlow applications. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 127–140, 2012.

[24] Cheng-Shang Chang, Wen-Jyh Chen, and Hsiang-Yi Huang. Birkhoff-von Neumann input buffered crossbar switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications*, volume 3, pages 1614–1623. IEEE, 2000.

[25] Cheng-Shang Chang, Wen-Jyh Chen, and Hsiang-Yi Huang. Birkhoff-von neumann input buffered crossbar switches. In *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*, volume 3, pages 1614–1623 vol.3, 2000.

[26] Feng Chen and Grigore Roşu. Java-MOP: A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, page 546–550, 2005.

[27] P4 Language Consortium. P4Lang Tutorials, Fetched July 15th, 2023. `https://github.com/p4lang/tutorials`.

[28] Giuseppe De Giacomo and Moshe Y. Vardi. Linear temporal logic and linear dynamic logic on finite traces. In *International Joint Conference on Artificial Intelligence (IJCAI)*, page 854–860, 2013.

[29] Stéphane Demri and Ranko Lazić. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic (TOCL)*, apr 2009.

[30] Dragos Dumitrescu, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. bf4: Towards bug-free P4 programs. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 571–585, 2020.

[31] Seyed K. Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. Efficient network reachability analysis using a succinct control plane representation. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, page 217–232, 2016.

[32] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022.

[33] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, page 469–483, 2015.

[34] Nate Foster, Dexter Kozen, Matthew Milano, Alexandra Silva, and Laure Thompson. A coalgebraic decision procedure for NetKAT. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 343–355, 2015.

[35] Open Networking Foundation. Aether: An open source 5G connected edge platform, 2022. `https://opennetworking.org/aether/`.

[36] Open Networking Foundation. Software Defined Fabric (SD-Fabric) Release 1.2, Fetched February 15th, 2023. `https://docs.sd-fabric.org/master/release/1.2.0.html`.

[37] Python Software Foundation. unittest.mock — Mock object library, 2025. `https://docs.python.org/3/library/unittest.mock.html`.

[38] Lucas Freire, Miguel Neves, Lucas Leal, Kirill Levchenko, Alberto Schaeffer-Filho, and Marinho Barcellos. Uncovering bugs in P4 programs with assertion-based verification. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[39] Swapnil Gandhi and Christos Kozyrakis. Moetion: Efficient and reliable checkpointing for mixture-of-experts models at scale, 2024.

[40] Swapnil Gandhi, Mark Zhao, Athinagoras Skiadopoulos, and Christos Kozyrakis. Recycle: Resilient training of large dnns using pipeline adaptation. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, SOSP '24, page 211–228, New York, NY, USA, 2024. Association for Computing Machinery.

[41] Adithya Gangidi, Rui Miao, Shengbao Zheng, Sai Jayesh Bondu, Guilherme Goes, Hany Morsy, Rohit Puri, Mohammad Riftadi, Ashmitha Jeevaraj Shetty, Jingyi Yang, Shuqiang Zhang, Mikel Jimenez Fernandez, Shashidhar Gandham, and Hongyi Zeng. Rdma over ethernet for distributed training at meta scale. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 57–70, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. Fast control plane analysis using an abstract representation. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 300–313, 2016.

[43] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in o(n log n) time in regular bipartite graphs. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, page 39–46, New York, NY, USA, 2010. Association for Computing Machinery.

[44] William Sealey Gosset. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[45] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[46] P. Hall. On representatives of subsets. *Journal of the London Mathematical Society*, s1-10(1):26–30, 1935.

[47] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, David Mazières, and Nick McKeown. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 71–85, 2014.

[48] John E Hopcroft and Richard M Karp. An nˆ5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[49] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism, 2019.

[50] Changho Hwang, Wei Cui, Yifan Xiong, Ziyue Yang, Ze Liu, Han Hu, Zilong Wang, Rafael Salas, Jithin Jose, Prabhat Ram, et al. Tutel: Adaptive mixture-of-experts at scale. *Proceedings of Machine Learning and Systems*, 5:269–287, 2023.

[51] Hyperopt. Hyperopt: Distributed Hyperparameter Optimization, Fetched May 21st, 2025. `https://github.com/hyperopt/hyperopt`.

[52] Intel. Intel® Tofino™., Fetched February 10th, 2023. `https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html`.

[53] Insu Jang, Zhenning Yang, Zhen Zhang, Xin Jin, and Mosharaf Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 382–395, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Karthick Jayaraman, Nikolaj Bjørner, Jitu Padhye, Amar Agrawal, Ashish Bhargava, Paul-Andre C Bissonnette, Shane Foster, Andrew Helwer, Mark Kasten, Ivan Lee, Anup Namdhari, Haseeb Niaz, Aniruddha Parkhi, Hanukumar Pinnamraju, Adrian Power, Neha Milind Raje, and Parag Sharma. Validating datacenters at scale. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 200–213, 2019.

[55] Chenyu Jiang, Ye Tian, Zhen Jia, Shuai Zheng, Chuan Wu, and Yida Wang. Lancet: Accelerating mixture-of-experts training via whole graph computation-communication overlapping, 2024.

[56] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. Tpu v4: An optically reconfigurable supercomputer for machine

learning with hardware support for embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[58] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 99–112, 2013.

[59] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 113–126, 2012.

[60] Anurag Khandelwal, Rachit Agarwal, and Ion Stoica. Confluo: Distributed monitoring and diagnosis stack for high-speed networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 421–436, 2019.

[61] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 15–27, 2013.

[62] Hyojoon Kim, Xiaoqi Chen, Jack Brassil, and Jennifer Rexford. Experience-driven research on programmable networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 51(1):10–17, January 2021.

[63] Hyojoon Kim and Arpit Gupta. Ontas: Flexible and scalable online network traffic anonymization system. In *ACM SIGCOMM Workshop on Network Meets AI & ML (NetAI)*, pages 15–21, 2019.

[64] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. *Proceedings of Machine Learning and Systems*, 5:341–353, 2023.

[65] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[66] K Shiv Kumar, K Ranjitha, PS Prashanth, Mina Tahmasbi Arashloo, U Venkanna, and Praveen Tammana. DBVal: Validating P4 data plane runtime behavior. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2021.

[67] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for Software-Defined Networks. In *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets)*, 2010.

[68] Haoyang Li, Fangcheng Fu, Hao Ge, Sheng Lin, Xuanyu Wang, Jiawen Niu, Yujie Wang, Hailin Zhang, Xiaonan Nie, and Bin Cui. Malleus: Straggler-resilient hybrid parallel training of large-scale models via malleable data and model parallelization, 2025.

[69] Wenxue Li, Xiangzhou Liu, Yuxuan Li, Yilun Jin, Han Tian, Zhizhen Zhong, Guyue Liu, Ying Zhang, and Kai Chen. Understanding communication characteristics of distributed training. In *Proceedings of the 8th Asia-Pacific Workshop on Networking*, APNet '24, page 1–8, New York, NY, USA, 2024. Association for Computing Machinery.

[70] Jinkun Lin, Ziheng Jiang, Zuquan Song, Sida Zhao, Menghan Yu, Zhanghan Wang, Chenyuan Wang, Zuocheng Shi, Xiang Shi, Wei Jia, Zherui Liu, Shuguang Wang, Haibin Lin, Xin Liu, Aurojit Panda, and Jinyang Li. Understanding stragglers in large model training using what-if analysis, 2025.

[71] Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context, 2023.

[72] Hong Liu, Ryohei Urata, Kevin Yasumura, Xiang Zhou, Roy Bannon, Jill Berger, Pedram Dashti, Norm Jouppi, Cedric Lam, Sheng Li, Erji Mao, Daniel Nelson, George Papen, Mukarram Tariq, and Amin Vahdat. Lightwave fabrics: At-scale optical circuit switching for datacenter and machine learning systems. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 499–515, New York, NY, USA, 2023. Association for Computing Machinery.

[73] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. P4v: Practical verification for programmable data planes. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 490–503, 2018.

[74] Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, page 499–512, 2015.

[75] Robert MacDavid, Carmelo Cascone, Pingping Lin, Badhrinath Padmanabhan, Ajay ThakuR, Larry Peterson, Jennifer Rexford, and Oguz Sunay. A P4-based 5G user plane function. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, page 162–168, 2021.

[76] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. Debugging the data plane with Anteater. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 290–301, 2011.

[77] N. McKeown. The islip scheduling algorithm for input-queued switches. *IEEE/ACM Transactions on Networking*, 7(2):188–201, 1999.

[78] Meta. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation, Fetched May 21st, 2025. https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

[79] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *ACM SIGCOMM Internet Measurement Conference (IMC)*, page 393–407, 2018.

[80] Microsoft. Microsoft Collective Communication Library, Fetched May 21st, 2025. https://github.com/microsoft/msccl.

[81] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[82] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm, 2021.

[83] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. P4Pktgen: Automated test case generation for P4 programs. In *ACM SIGCOMM Symposium on SDN Research (SOSR)*, 2018.

[84] NVIDIA. A New Era in Data Center Networking with NVIDIA Silicon Photonics-based Network Switching, 2025. https://developer.nvidia.com/blog/a-new-era-in-data-center-networking-with-nvidia-silicon-photonics-based-network-switching/.

[85] NVIDIA. Megatron-LM repository, Fetched April 16th, 2025. https://github.com/NVIDIA/Megatron-LM.

[86] NVIDIA. Collective operations, Fetched March 31st, 2025. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html.

[87] NVIDIA. NVIDIA NeMo Framework User Guide: Communication Overlap, Fetched March 31st, 2025. https://docs.nvidia.com/nemo-framework/user-guide/latest/nemotoolkit/features/optimizations/communication_overlap.html.

[88] NVIDIA. NVIDIA Quantum InfiniBand Switches, Fetched May 6th, 2025. https://www.nvidia.com/en-us/networking/infiniband-switching/.

[89] NVIDIA. NVLink and NVLink Switch, Fetched May 6th, 2025. https://www.nvidia.com/en-us/data-center/nvlink/.

[90] Giannis Patronas, Nikos Terzenidis, Prethvi Kashinkunti, Eitan Zahavi, Dimitris Syrivelis, Louis Capps, Zsolt-Alon Wertheimer, Nikos Argyris, Athanasios Fevgas, Craig Thompson, Avraham Ganor, Julie Bernauer, Elad Mentovich, and Paraskevas Bakopoulos. Optical switching for data centers and advanced computing systems. *J. Opt. Commun. Netw.*, 17(1):A87–A95, Jan 2025.

[91] Polatis. Huber+Suhner Polatis, Fetched November 4th, 2025. `https://www.polatis.com/`.

[92] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. *SIGCOMM Comput. Commun. Rev.*, 43(4):447–458, August 2013.

[93] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating microsecond circuit switching into the data center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, page 447–458, New York, NY, USA, 2013. Association for Computing Machinery.

[94] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 66–85, New York, NY, USA, 2022. Association for Computing Machinery.

[95] PyTorch. Meta device, Fetched April 16th, 2025. `https://pytorch.org/docs/stable/meta.html`.

[96] Kun Qian, Yongqing Xi, Jiamin Cao, Jiaqi Gao, Yichi Xu, Yu Guan, Binzhang Fu, Xuemei Shi, Fangbo Zhu, Rui Miao, Chao Wang, Peng Wang, Pengcheng Zhang, Xianlong Zeng, Eddie Ruan, Zhiping Yao, Ennan Zhai, and Dennis Cai. Alibaba hpn: A data center network for large language model training. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 691–706, New York, NY, USA, 2024. Association for Computing Machinery.

[97] Sudarsanan Rajasekaran, Manya Ghobadi, and Aditya Akella. {CASSINI}:{Network-Aware} job scheduling in machine learning clusters. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1403–1420, 2024.

[98] Sudarsanan Rajasekaran, Sanjoli Narang, Anton A. Zabreyko, and Manya Ghobadi. Mltcp: A distributed technique to approximate centralized flow scheduling for machine learning. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks*, HotNets '24, page 167–176, New York, NY, USA, 2024. Association for Computing Machinery.

[99] Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International conference on machine learning*, pages 18332–18346. PMLR, 2022.

[100] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models, 2020.

[101] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker. Abstractions for network update. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 323–334, 2012.

[102] Sundararajan Renganathan. Genstack: A Megatron-LM based communication tracer and permutation generator, 2025. `https://github.com/sundararajan/genstack`.

[103] Sundararajan Renganathan and Nick McKeown. Genstack: A Megatron-LM based communication tracer and permutation generator, 2025. `https://github.com/sundararajan20/genstack/`.

[104] Sundararajan Renganathan, Benny Rubin, Hyojoon Kim, Pier Luigi Ventre, Carmelo Cascone, Daniele Moro, Charles Chan, Nick McKeown, and Nate Foster. Hydra: Effective network verification (full version), September 2023. Available at `https://www.cs.cornell.edu/~jnfoster/papers/hydra-tr.pdf`.

[105] Databricks Mosaic Research. Composer repository, Fetched May 6th, 2025. `https://github.com/mosaicml/composer`.

[106] Fabian Ruffy, Jed Liu, Prathima Kotikalapudi, Vojtěch Havel, Hanneli Tavante, Rob Sherwood, Vlad Dubina, Volodymyr Peschanenko, Anirudh Sivaraman, and Nate Foster. P4Testgen: An extensible test oracle for P4. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, 2023. To appear.

[107] R. Ryf, J. Kim, J.P. Hickey, A. Gnauck, D. Carr, F. Pardo, C. Bolle, R. Frahm, N. Basavanhally, C. Yoh, D. Ramsey, R. Boie, R. George, J. Kraus, C. Lichtenwalner, R. Papazian, J. Gates, H.R. Shea, A. Gasparyan, V. Muratov, J.E. Griffith, J.A. Prybyla, S. Goyal, C.D. White, M.T. Lin, R. Ruel, C. Nijander, S. Arney, D.T. Neilson, D.J. Bishop, P. Kolodner, S. Pau, C. Nuzman, A. Weis, B. Kumar, D. Lieuwen, V. Aksyuk, D.S. Greywall, T.C. Lee, H.T.

Soh, W.M. Mansfield, S. Jin, W.Y. Lai, H.A. Huggins, D.L. Barr, R.A. Cirelli, G.R. Bogart, K. Teffeau, R. Vella, H. Mavoori, A. Ramirez, N.A. Ciampa, F.P. Klemens, M.D. Morris, T. Boone, J.Q. Liu, J.M. Rosamilia, and C.R. Giles. 1296-port mems transparent optical crossconnect with 2.07 petabit/s switch capacity. In *OFC 2001. Optical Fiber Communication Conference and Exhibit. Technical Digest Postconference Edition (IEEE Cat. 01CH37171)*, volume 4, pages PD28–PD28, 2001.

[108] Daniele De Sensi, Tommaso Bonato, David Saam, and Torsten Hoefler. Swing: Short-cutting rings for higher bandwidth allreduce. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1445–1462, Santa Clara, CA, April 2024. USENIX Association.

[109] Amazon Web Services. AWS Trainium, Fetched May 6th, 2025. `https://aws.amazon.com/ai/machine-learning/trainium/`.

[110] Aashaka Shah, Vijay Chidambaram, Meghan Cowan, Saeed Maleki, Madan Musuvathi, Todd Mytkowicz, Jacob Nelson, Olli Saarikivi, and Rachee Singh. TACCL: Guiding collective algorithm synthesis using communication sketches. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 593–612, Boston, MA, April 2023. USENIX Association.

[111] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[112] Apoorv Shukla, Seifeddine Fathalli, Thomas Zinner, Artur Hecker, and Stefan Schmid. P4consist: Toward consistent P4 SDNs. *IEEE Journal on Selected Areas in Communications (JSAC)*, 38(7):1293–1307, 2020.

[113] Alan Smith and Vamsi Krishna Alla. Amd instinct™ mi300x: A generative ai accelerator and platform architecture. *IEEE Micro*, pages 1–9, 2025.

[114] Wilson Snyder, Paul Wasson, Duane Galbi, and et al. Verilator.

[115] S. Sohma, T. Watanabe, T. Shibata, and H. Takahashi. Compact and low power consumption 16 /spl times/ 16 optical matrix switch with silica-based plc technology. In *OFC/NFOEC Technical Digest. Optical Fiber Communication Conference, 2005.*, volume 4, pages 3 pp. Vol. 4–, 2005.

[116] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Debugging P4 programs with Vera. In *ACM Special Interest Group on Data Communication (SIGCOMM)*, page 518–532, 2018.

[117] Yuval Tamir and H-C Chi. Symmetric crossbar arbiters for vlsi communication switches. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):13–27, 1993.

[118] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Simplifying datacenter network debugging with Pathdump. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, page 233–248, 2016.

[119] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. Distributed network monitoring and debugging with SwitchPointer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 453–456, 2018.

[120] Telescent. Telescent G5 NTM, Fetched November 4th, 2025. `https://www.telescent.com/products`.

[121] John Thorpe, Pengzhan Zhao, Jonathan Eyolfson, Yifan Qiao, Zhihao Jia, Minjia Zhang, Ravi Netravali, and Guoqing Harry Xu. Bamboo: Making preemptible instances resilient for affordable training of large DNNs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 497–513, Boston, MA, April 2023. USENIX Association.

[122] Bingchuan Tian, Jiaqi Gao, Mengqi Liu, Ennan Zhai, Yanqing Chen, Yu Zhou, Li Dai, Feng Yan, Mengjing Ma, Ming Tang, Jie Lu, Xionglie Wei, Hongqiang Harry Liu, Ming Zhang, Chen Tian, and Minlan Yu. Aquila: A practically usable verification system for production-scale programmable data planes. In *ACM Special Interest Group on Data Communications (SIGCOMM)*, page 17–32, 2021.

[123] UEC. Ultra Ethernet Consortium, Fetched May 6th, 2025. `https://ultraethernet.org/`.

[124] Ryohei Urata, Hong Liu, Kevin Yasumura, Erji Mao, Jill Berger, Xiang Zhou, Cedric Lam, Roy Bannon, Darren Hutchinson, Daniel Nelson, Leon Poutievski, Arjun Singh, Joon Ong, and Amin Vahdat. Mission apollo: Landing optical circuit switching at datacenter scale, 2022.

[125] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C. Snoeren. Practical tdma for datacenter ethernet. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 225–238, New York, NY, USA, 2012. Association for Computing Machinery.

[126] Vijay V. Vazirani. *Approximation algorithms*. Springer-Verlag, Berlin, Heidelberg, 2001.

[127] John Von Neumann. A certain zero-sum two-person game equivalent to the optimal assignment problem. *Contributions to the Theory of Games*, 2(0):5–12, 1953.

[128] Hao Wang, Han Tian, Jingrong Chen, Xinchen Wan, Jiacheng Xia, Gaoxiong Zeng, Wei Bai, Junchen Jiang, Yong Wang, and Kai Chen. Towards Domain-Specific network transport

for distributed DNN training. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1421–1443, Santa Clara, CA, April 2024. USENIX Association.

[129] Weiyang Wang, Moein Khazraee, Zhizhen Zhong, Manya Ghobadi, Zhihao Jia, Dheevatsa Mudigere, Ying Zhang, and Anthony Kewitsch. TopoOpt: Co-optimizing network topology and parallelization strategy for distributed training jobs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 739–767, Boston, MA, April 2023. USENIX Association.

[130] Ertza Warraich, Omer Shabtai, Khalid Manaa, Shay Vargaftik, Yonatan Piasetzky, Matty Kadosh, Lalith Suresh, and Muhammad Shahbaz. OptiReduce: Resilient and Tail-Optimal AllReduce for distributed deep learning in the cloud. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 685–703, Philadelphia, PA, April 2025. USENIX Association.

[131] Tianyuan Wu, Wei Wang, Yinghao Yu, Siran Yang, Wenchao Wu, Qinkai Duan, Guodong Yang, Jiamang Wang, Lin Qu, and Liping Zhang. Falcon: Pinpointing and mitigating stragglers for large-scale hybrid-parallel training, 2024.

[132] Xin Wu, Daniel Turner, Chao-Chih Chen, David A. Maltz, Xiaowei Yang, Lihua Yuan, and Ming Zhang. Netpilot: Automating datacenter network failure mitigation. *ACM SIGCOMM Computer Communication Review (CCR)*, 42(4):419–430, August 2012.

[133] Yongji Wu, Wenjie Qu, Tianyang Tao, Zhuang Wang, Wei Bai, Zhuohao Li, Yuan Tian, Jiaheng Zhang, Matthew Lentz, and Danyang Zhuo. Lazarus: Resilient and elastic training of mixture-of-experts models with adaptive expert placement, 2024.

[134] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. On static reachability analysis of IP networks. In *IEEE Conference on Computer Communications (INFOCOM)*, pages 2170–2183, 2005.

[135] Jinjun Xiong, Yiu-Chung Wong, Egino Sarto, and Lei He. Constraint driven i/o planning and placement for chip-package co-design. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, pages 207–212, 2006.

[136] Hongkun Yang and Simon S Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, 2015.

[137] Hongkun Yang and Simon S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 24(2):887–900, April 2016.

[138] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 701–718, 2020.

[139] S. J. Ben Yoo. Optical packet and burst switching technologies for the future photonic internet. *Journal of Lightwave Technology*, 24(12):4468–4492, 2006.

[140] Hongyi Zeng, Peyman Kazemian, George Varghese, and Nick McKeown. Automatic test packet generation. In *Emerging Networking EXperiments and Technologies (CoNEXT)*, page 241–252, 2012.

[141] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, page 87–99, 2014.

[142] Kaiyuan Zhang, Danyang Zhuo, Aditya Akella, Arvind Krishnamurthy, and Xi Wang. Automated verification of customizable middlebox properties with gravel. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 221–239, 2020.

[143] Peng Zhang, Hao Li, Chengchen Hu, Liujia Hu, Lei Xiong, Ruilong Wang, and Yuemei Zhang. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Emerging Networking EXperiments and Technologies (CoNEXT)*, page 19–33, 2016.

[144] Liangyu Zhao, Siddharth Pal, Tapan Chugh, Weiyang Wang, Jason Fantl, Prithwish Basu, Joud Khoury, and Arvind Krishnamurthy. Efficient Direct-Connect topologies for collective communications. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, pages 705–737, Philadelphia, PA, April 2025. USENIX Association.

[145] Yihao Zhao, Yuanqiang Liu, Yanghua Peng, Yibo Zhu, Xuanzhe Liu, and Xin Jin. Multi-resource interleaving for deep learning training. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 428–440, New York, NY, USA, 2022. Association for Computing Machinery.

[146] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.