Mechanisms to Advance the Adoption of Programmable High-speed Packet-Processing Pipelines

by

Tao Wang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

May, 2025

Dr. Anirudh Sivaraman

Dr. Aurojit Panda

Dedication

To all the ones who helped me survive.

Acknowledgements

When I reflect on my past six-year journey toward a Ph.D., I realize that the most valuable asset I have is not the challenging projects I have done but the precious friendships with so many individuals along the journey. It would not have been possible without the support, guidance, and encouragement of them, to whom I am deeply grateful.

First and foremost, I would like to express my sincere gratitude to my advisors for their invaluable guidance, patience, and unwavering support during my journey.

Aurojit Panda: As an advisor, he is omniscient (I will not question that) about everything. His hands-on experience in building and optimizing systems always guides us in the right direction through the challenges. As a friend, you can easily feel how energetic he is about the research projects. I still remember he was introducing his friends to me and raising a thumbs-up high in the crowd after my presentation of Menshen at NSDI'22, which really gave me a sense that you are not alone in this challenging and exhausting journey. Furthermore, life is not a place full of candy, Panda always gives different perspectives to help you see through murky things. If possible, I would consult him for advice throughout my life.

Anirudh Sivaraman: Anirudh, as my advisor has deeply shaped my research taste during my early years at NYU. I still remember he told me that we should be careful about picking the topics we really want to work on, since we only have limited time (well, maybe I understood it in a wrong way that I so many times directly rejected the early ideas—which were shown to be worth doing by other groups—proposed by him). More importantly to me, a usually pessimistic guy from Wuhan,

Anirudh had done so many mental massages during the pandemic, which gave me a lot of relief. I would say a sincere thank you (in the form of a message rather than a direct phone call, which really drove me nervous in my early years).

Gianni Antichi: As the Chinese proverb says, an elderly person in the family is like a treasure. Gianni is such an "elderly" person during my journey. He is always ready and willing to help even for the tiny bugs of handshaking in Verilog. His experience in FPGA development helped me a lot when we started both Menshen and QingNiao projects. Most importantly, I can sense the energy and enthusiasm about the research itself when working with him. If allowed, I would someday join his paper reading group at a random bar drinking beers with him and Panda :-).

I am also deeply thankful to my other dissertation committee members Jinyang Li and Ratul Mahajan for their time, thoughtful critiques, and constructive feedback, which have greatly improved the quality of my dissertation.

Another heartfelt thank you goes to my friends who I had collaborated with closely.

Fabian Ruffy: During the collaboration with him on the Gauntlet project, I found Fabian is phenomenally hard-working and skilled at team management. But most importantly, he aims super high for all his projects, deciding to build practical tools that can be used by the broader developers. All his attitudes influenced a lot me in my own projects.

Xiangrui Yang: Xiangrui is literally the first guy who taught me how to program in Verilog for a big project. I still remember the days and nights that I worked before the computer waiting for the synthesis results and kept improving my implementation with him. Interestingly, this experience gives me a chance to practice the Pomodoro technique of time management.

Jinkun Lin: Jinkun is an optimistic guy who helped me survive the "potential" mental breakdown when we were doing the "bumpy" QingNiao project. We took a holistic approach which involves modifying everything from the application to the hardware in QingNiao, which is challenging and daunting in terms of the engineering effort. I still remember Jinkun's warm words and optimistic attitudes along the journey we developed QingNiao.

Hang Zhu: Hang is an insightful researcher in the networking field. It was an interesting opportunity for me to collaborate with Hang, since we have similar visions but different angles to tackle the problem of how to support multitenancy for programmable networks. The collaboration with Hang taught me that we should not always consider things from one specific angle. The software world is also amazing.

Alex Forencich: Alex, an unselfish researcher from UCSD, has maintained the repository of Corundum, which I have used throughout my research. He has devoted himself to the open-source community of FPGA development and taught me how important it is to be enthusiastic about the things you are doing, especially in the field of developing hardware. To make the prototype on FPGA happen needs consistent effort on improving the every low-level details of your implementations.

I would also like to acknowledge all the support from my friends and colleagues in NYU NetSys group including **Cheng Tan**, **Lingfan Yu**, **Changgeng Zhao**, **Yu Cao**, **Xiangyu Gao**, **Anqi Zhang**, **Jiaxin Lin**, **Qiongwen Xu**, **Zhanghan Wang** and many others.

Last, but not least, I probably would not have gone this far without the tremendous support from my parents. Although I always have different or even opposite perspectives with them, they always support my decisions and hope the best for me in the end without hesitation.

Abstract

Modern high-speed programmable packet-processing pipelines—typically built on the Reconfigurable Match Table (RMT) architecture—have enabled a wide range of network offloads, such as in-network telemetry, caching, and machine learning parameter aggregation. However, these capabilities remain largely inaccessible to a broader range of users and applications.

This dissertation investigates the barriers to broader adoption from two key perspectives: multitenancy and general application-level, *i.e.,* Layer 7 (L7), processing. It argues that expanding adoption requires new hardware primitives alongside complementary software toolchains. To address these challenges, the dissertation introduces two systems: (1) Menshen, which provides isolation mechanisms that allow multiple programs to safely share a single pipeline without interfering with each other; and (2) QingNiao, a system designed for L7 dispatch—a pervasive and crucial L7 processing in network infrastructure—offering a holistic solution that combines novel hardware architecture design with a high-level programming model to enable efficient L7 dispatch on programmable pipelines.

Contents

List of Figures

List of Tables

Previously Published Materials

Chapter 3 combines materials from two previous publications [135, 136]:

- Tao Wang*, Hang Zhu*, Fabian Ruffy, Xin Jin, Anirudh Sivaraman, Dan Ports, and Aurojit Panda. Multitenancy for Fast and Programmable Networks in the Cloud. In USENIX HotCloud, 2020. (*Co-primary authors)

- Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. Isolation Mechanisms for High-speed Packet-processing Pipelines. In USENIX NSDI, 2022.

Chapter 4 adapts materials from the following arXiv paper [134]:

- Tao Wang, Jinkun Lin, Gianni Antichi, Aurojit Panda, and Anirudh Sivaraman. Application-Defined Receive Side Dispatching on the NIC. In arXiv, 2024.

1 | Introduction

Programmable high-speed packet-processing pipelines—usually integrated in networking devices like switches/routers [22] and Smart Network Interface Cards (SmartNICs [35, 2])—allow the data center network infrastructure to provide additional features beyond packet forwarding/routing. Recent work has demonstrated that offloading functionalities such as in-network caching [100], consensus protocols [99, 107], congestion control based on in-network telemetry [108], monitoring [71], etc., to these programmable networking devices can improve throughput and reduce latency, thus freeing up precious CPU cycles [82].

However, the benefits of these programmable networking devices[1] are not yet broadly accessible to a wider range of users and applications.

One major barrier is **multitenancy**: the ability for **multiple** independently developed network offloads to simultaneously run atop a **single** programmable high-speed packet-processing pipeline without interfering with each other.

Most existing systems assume that a single offload occupies the entire programmable pipeline in a fully dedicated setting. However, as network programmability matures, there is growing demand for devices that can multiplex offloads to conserve hardware resources. For example, public cloud may allow tenants to install their own offloads (*e.g.,* in-network caching [100], telemetry [71]) on the cloud provider's devices. Similarly, within a single organization, multiple teams may need

[1] Throughout this dissertation, we focus our discussion on Application-Specific Integrated Circuit (ASIC) based high-speed packet-processing pipelines that are typically built on a Reconfigurable Match Table (RMT) architecture, as detailed in Chapter 2.

to colocate different offloads, *e.g.,* an in-network measurement, and an in-network aggregation module for machine-learning [126].

Another barrier is support for general application-level, *i.e.,* Layer 7 (L7), processing. Most existing systems' offloads only work on per-packet network headers (*e.g.,* TCP/IP [47]), which have fixed-length fields. However, L7 messages can be highly variable as application developers can define their own messages [23, 43]. For instance, a web server may first extract the `path` field from the HTTP header [46] and then respond with appropriate contents. The `path` field can vary in length, and sometimes extends to tens of bytes exceeding the parsing capacity of the current programmable pipelines as detailed in §2.

Unfortunately, today's ASIC-based high-speed programmable packet-processing pipelines—commonly built on the Reconfigurable Match Table (RMT) [73] architecture, as seen in commercial products like Tofino [22] switch, Intel Mount Evans IPU [21], Pensando SmartNIC [2]—do not adequately support the following:

- **Multitenancy.** They lack the isolation mechanisms to simultaneously run multiple offloads safely without interfering with each other, like accessing allocated hardware resources or achieving non-degrading performance.

- **General L7 processing.** They cannot parse or act on variable-length L7 fields located at arbitrary offsets in packet payloads, making it infeasible to offload application-level functionalities that operate on such variable-length L7 fields.

In this dissertation, we present the design and implementation of two systems, *i.e.,* Menshen and QingNiao, to augment existing programmable pipelines with the support of multitenancy and L7 processing, respectively. Our systems are based on the observation that:

Thesis Statement. To enable broader adoption of the programmable pipelines among application developers—particularly in shared environments like public clouds—mechanisms that combine new hardware primitives with supporting software toolchains are essential. These mecha-

nisms must make programmable pipelines more accessible and provide application-defined offloads with strong performance improvements and isolation guarantees.

The remainder of this dissertation is organized as follows: We provide the background information in Chapter 2 and focus on the discussion of the barriers hindering existing programmable pipelines from being widely adopted. Then in Chapter 3, we present Menshen, a system that provides isolation mechanisms for running multiple offloads on a single programmable pipeline. Next in Chapter 4, we present QingNiao, a framework for the developers to harvest the benefits of the programmable pipeline by specifically enabling offloading the functionality of L7 dispatch. Finally, we conclude by describing limitations and future work in Chapter 5.

2 | Background

We begin by introducing the detailed architecture of the ASIC-based high-speed packet-processing pipeline, *i.e.,* Reconfigurable Match Table [73] (RMT) architecture which is widely recognized in both academia (*e.g.,* research prototypes [100, 99, 107] are built atop, etc.) and industry (*e.g.,* commercial products, like Intel Tofino switches [22] and Pensando SmartNICs [2], are released). We then elaborate on the reasons why such RMT-based packet-processing pipelines are incapable of supporting multitenancy and general L7 processing.

2.1 Reconfigurable Match Table

Programmable networking devices fall into three main categories: (1) Application-Specific Integrated Circuit (ASIC) based high-speed packet-processing pipelines *e.g.,* Intel Tofino switch [22], Pensando SmartNIC [2]; (2) Field Programmable Gate Arrays (FPGA) based devices like Cisco Nexus SmartNIC [8]; (3) Multicore System on a Chip (SoC) based devices like Marvell LiquidIO SmartNIC [28]. Each offers a different tradeoff between performance (*e.g.,* device throughput, clock speed, power consumption, etc.) and flexibility (*e.g.,* programmability). Among these, ASIC-based devices provide the highest performance potential compared to the other two [82], which is crucial as networks scale to 400Gbps and beyond. Thus, in this dissertation, we focus on ASIC-based high-speed packet-processing pipelines using the RMT architecture [73] hereafter.

Architectural overview. As depicted in Figure 2.1, an RMT pipeline typically comprises a pro-

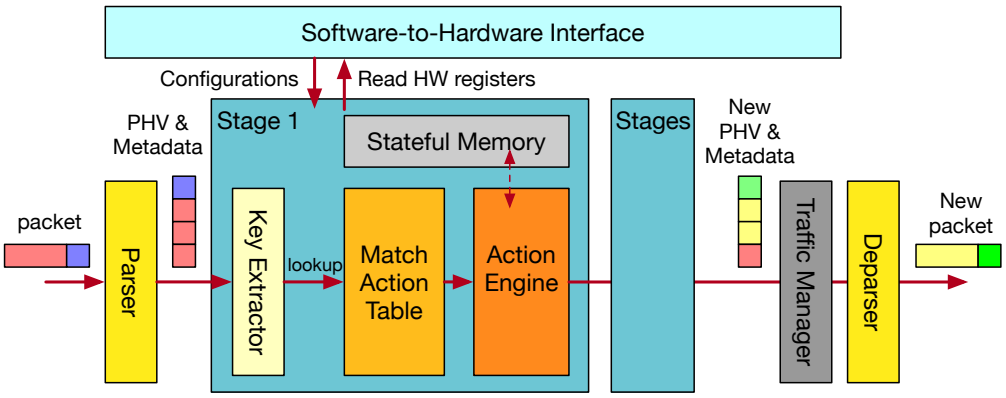


Figure 2.1: Example of an RMT pipeline.

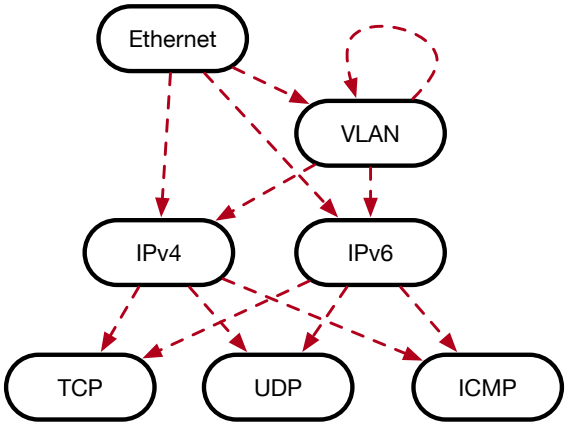


Figure 2.2: Example of a parser graph. The nodes represent the parsing states, while the arrows represent the transitions between parsing states.

grammable parser, multiple stages, a traffic manager, and a final programmable deparser at the end. Except for the traffic manager—which is used for packet scheduling—, other elements are programmable and exposed to the developer: typically, a compiler [39] is used to convert the application offload/program[1] into the configurations, which are installed on the pipeline via a software-to-hardware interface.

Programmable parser. At a high level, the parser of each offload is represented by a parser graph shown in Figure 2.2. When a packet comes in, the parser will serially traverse through the packet's byte stream according to the parser graph to match the nodes/states configured, and extract the

[1]We use offload and program interchangeably in this dissertation. Both terms represent the parts of an application that the developer intends to offload to the hardware.

related header fields. For example, as Figure 2.2 shows, the parser graph will first check whether it is an Ethernet packet and then move along the edges/transitions to reach the final nodes/states to see whether the packet is a TCP, UDP, or ICMP packet. In this way, the parser extracts the header fields (*e.g.,* IP source and destination addresses, etc.)—usually program-specific—from the packet and stores them in the Packet Header Vectors (PHVs). Along with the pipeline metadata (*e.g.,* the port where the packet is received, etc.), PHVs are passed to the following consecutive stages. Such a programmable parser is usually implemented using a Ternary Content Addressable Memory (TCAM) and a Static Random Access Memory (SRAM) as discussed in [89].

Programmable stages. Each stage forms keys out of headers, looks up the keys in a match-action table, and then performs the matched actions. Specifically, within a stage, upon receiving PHVs with the metadata, a key extractor is configured in a per-program manner to combine fields from the PHVs and the metadata to generate the key. Then, this key is compared against the entries in the match-action table for the associated action, which is carried out in the action engine, and the results are written back to the PHVs and passed to the next stage. For a concrete example, to fulfill the functionality of a Network Address Translator (NAT), both the packet's IP source and destination addresses are kept in the PHVs. A stage is configured to match against the incoming packet's IP source address and modify its destination IP address accordingly based on the address translation table configured.

Programmable deparser. After all processing stages, deparser assembles the modified PHVs back to the packet according to the parser graph configured, which is the reverse path of the parser, and emits the packet out from the pipeline.

2.2 RMT's Limitations for Multitenancy

As described, RMT allows compiling offloads into pipeline configurations that can be directly loaded into the pipeline. It does not natively support running multiple independently developed

offloads on a single pipeline. There are several key limitations:

- It lacks the interface to coordinate different offloads. For example, two configurations of two application offloads might conflict with each other after compilation (*e.g.,* without coordination, they might be both packed into the same stage without enough resources), which forbids them from being successfully loaded on the same device.

- It lacks the interface to specify the hardware resources used by each offload. Although some annotations [39] are provided to indicate how to place offloads, it is challenging to fine-tune those annotations since they are too coarse-grained and may not be strictly enforced.

- It lacks the support of disruption-free reconfigurations. Suppose one developer wants to add features to its offload, it should be guaranteed that its reconfigurations do not impact the running offloads. However, existing RMT can not support disruption-free reconfiguration since it has to refresh the whole pipeline.

Inherently, the RMT architecture poses unique challenges for isolation because its pipeline design means that neither an OS nor a hypervisor can be used to enforce isolation in the data plane.[2] This is because RMT is a *dataflow* or spatial hardware architecture [68, 78] with a set of instructions units continuously processing data (packets). This is in contrast to the Von Neumann architecture found on processors [59], where a program counter decides what instruction to execute next. As such, an RMT pipeline is closer in its hardware architecture to an FPGA or a CGRA [118] than a processor. This difference in architecture has important implications for isolation. The Von Neumann architecture supports a *time-sharing* approach to isolation (in the form of an OS/hypervisor) that runs different programs on the CPU successively by changing the program counter to point to the next instruction of the next module, which can not be directly applied to the *dataflow* architecture like RMT.

[2] An OS does run on the network device's control CPU, allowing isolation in the control plane. Our focus is the isolation of the RMT pipeline itself.

To this end, in Chapter 3, we first lay out the requirements of multitenancy for RMT pipelines. Then, we propose a software-hardware codesign, called Menshen, which leverages *space-partitioning* to fulfill the desired requirements.

2.3 RMT's Limitations for General L7 Processing

Additionally, as discussed, RMT is inherently constrained by its design and its hardware resources, which limits its capability of processing general L7 fields. The philosophy of RMT is to (1) extract, (2) store, and (3) modify the related fields by using the PHVs, which means that:

- **Parsing complexity.** RMT is not able to host an offload if its parser graph is too complex to be loaded onto the pipeline.
- **PHV limitation.** RMT can not store the fields to the PHVs if the lengths of the fields used in the offload are excessively long.
- **Action limitation.** RMT can not provide the desired modifications to the fields if the operations—beyond addition, subtraction, etc.[135]—are too complicated.

However, this is always the case for the general L7 processing. Taking HTTP load balancer [34] as an example, it usually operates on the `path` field inside the HTTP header. The `path` field usually encapsulates the URLs that have variable lengths and start at arbitrary offsets of a packet. This makes it hard for RMT to process.

First, different from extracting header fields for L3/4 fields where the length is fixed, in order to extract the URLs, the developer can not specify the fixed size of such fields. Instead, the developer has to add a sufficient number of branches to cover all the length cases of the URLs, which will definitely cause a state explosion in the parser graph. Second, the numbers and sizes of PHVs in RMT are usually limited (*e.g.,* there are tens of PHVs in Tofino [22], each is at most 8 bytes), the RMT pipeline can not store the URLs in a PHV even if it can extract it out. Third, carrying out

actions on PHVs in each stage requires large crossbars [112], which are both resource-heavy and performance-sensitive. It might be impossible to fabricate such RMT pipelines with large crossbars to handle arbitrarily long fields like URLs. All these reasons make it challenging for the existing RMT architecture to support general L7 processing.

To this, in Chapter 4, we first analyze the essential reasons why the existing ASIC-based pipeline can not support processing general L7 fields. Then, we narrow down our scope to a specific L7 processing—*i.e.,* L7 dispatch, which is widely used in the service mesh consisting of software proxies—and propose our framework, called QingNiao, to address the issues.

3 | Menshen

This chapter presents Menshen, a software-hardware codesign that addresses one aspect of the aforementioned *accessibility*, *i.e.,* multitenancy. To simultaneously and safely run multiple programs on a single device, it is required to support isolation. However, existing high-speed packet-processing pipelines such as RMT provide only limited support for isolation as we discussed in Chapter 2. For example, it is possible to share stateful memory across programs but cannot share other resources like match-action tables [142]. We begin by laying out the requirements for isolation mechanisms on the RMT architecture that are applicable to all resources throughout the pipeline and then we detail the design and implementation of Menshen.

3.1 Requirements for Isolation Mechanisms on RMT

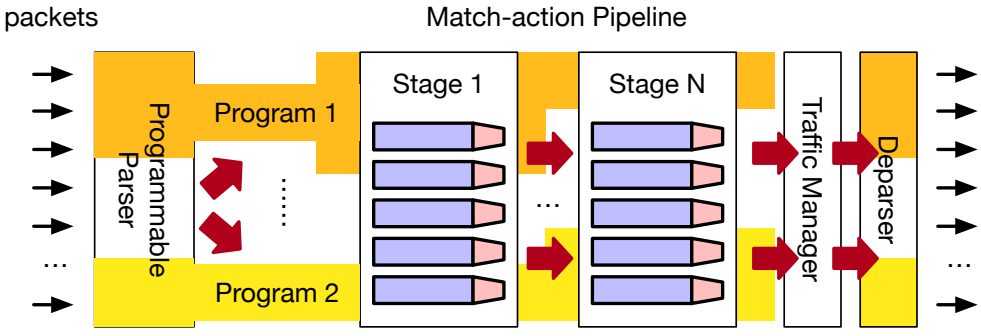


Figure 3.1: An example of running multiple programs on a single RMT pipeline. We show the resources allocated to different programs by shading them in the appropriate colors.

As presented in Figure 3.1, the desired isolation mechanisms should guarantee that multiple

programs can be allocated to different resources, and process packets in parallel without impacting each other. Specifically, isolation mechanisms should ensure that:

1. **Behavior isolation.** The behavior of one program must not affect the behavior (*i.e.,* input, output, computation, and internal state) of another. This would prevent a faulty or malicious program from adversely affecting other programs. Further, one program should not be able to inspect the behavior of another program.

2. **Resource isolation.** A switch/NIC pipeline has multiple resources, *e.g.,* static random-access memory (SRAM) for exact matching and ternary content-addressable memory (TCAM) for ternary matching. Each program should be able to access only its assigned subset of the pipeline's resources and no more. It should also be possible to allocate each resource independent of other resources. For example, an in-network caching program may need large amounts of stateful memory [100] for its caches, but a routing program may need significant TCAM for routing tables.

3. **Performance isolation.** Each program should stay within its allotted ingress packets per second and bits per second rates. One program's behavior should not affect the throughput and latency of another program.

4. **Lightweight.** The isolation mechanisms themselves must have low overhead so that their presence does not significantly degrade the high performance of the underlying network device. In addition, the extra hardware consumed by these mechanisms must be small.

5. **Rapid reconfiguration.** If a program is reconfigured with new packet-processing logic, the reconfiguration process should be quick.

6. **No disruption.** If a program is reconfigured, it must not disrupt the behavior of other unchanged programs—especially important in a multi-tenant environment [79].

3.2 Menshen's Approach

As discussed in Chapter 2, RMT is inherently a dataflow architecture, thus *time-sharing* approaches, widely used in the OS/hypervisor, can not be directly applied to RMT. This is mainly because swapping in/out the configurations for different programs to process each packet at the nanosecond level is impossible.

In order to meet its performance goals, RMT's pipelined architecture ensures that processing stages never stall, *i.e.,* they can process a packet every clock cycle. The Menshen design aims to preserve this invariant so that isolation does not come at the cost of performance. To maintain this invariant, Menshen's isolation mechanisms cannot reconfigure stages or change table contents between packets. As a result, Menshen provides isolation by *spatially partitioning* pipeline resources between packet processing programs.

While spatial partitioning is easy for resources, *e.g.,* match-action tables and stateful memory, that are provisioned so they can be allocated at a coarse granularity, it is much more challenging for resources such as key extractors (Chapter 2) which are generally shared across flows. This is because naive approaches to spatially partitioning such shared resources across packet-processing programs would severely reduce the number of resources available to each packet-processing program—and hence the richness and expressiveness of that program.

To see why, consider a case where a key extractor is split between two packet processing programs: in this setting each packet processing program can only use half the key extractor, limiting its key length to half of what it would be able to use were it running on the entire pipeline. Concretely, suppose the original one selects 6 PHVs, if it is evenly partitioned between 2 programs, only 3 PHVs can be selected by each program, which makes the original program that uses 6 PHVs to form keys impossible to run. This problem is of course further exacerbated as we increase the number of packet processing programs sharing the pipeline.

Menshen addresses this problem using *overlays*: we associate a configuration lookup table with

Applied Mechanism	Targeted Resources
Space partitioning	Match action table entries, stateful memories
Overlays	Parsing actions, key extractors, packet header vector (PHV) containers, arithmetic logic units (ALUs)

Table 3.1: Summary of Menshen's mechanisms.

each shared resource through the RMT pipeline. Menshen adds additional hardware primitives in the form of small tables that store program-specific configurations. As a packet progresses through the pipeline, the packet's program identifier is used as an index into these tables to extract program-specific configurations before processing the packet according to the just extracted configuration. These primitives are similar to the use of *overlays* [37, 4] in embedded systems [54, 1] and earlier PCs [38]. They effectively allow us to bring in different configurations for the same RMT resource, in response to different packets from different programs. For example, in the case of the key extractor, the configuration table contains the instructions that the program uses to construct the key (Chapter 2). Our use of overlays means that we do not need to partition resources including ALUs or PHVs between programs. Instead, the program has exclusive access to all PHVs/ALUs in a stage when processing a packet. Table 3.1 summarizes Menshen mechanisms.

3.3 Menshen's Design

At a high level, as shown in Figure 3.2, Menshen combines both the software toolchain with the proposed hardware primitives which are pervasively throughout the RMT pipeline. Specifically, Menshen software (§3.3.1) consists of (1) a system-level program; (2) a compiler that supports resource sharing policy, a resource usage checker, and a sanity checker with static analysis; (3) a software-to-hardware interface like P4Runtime [40], which is used to interact with the underlying hardware pipeline, *e.g.,* configure table entries, gather statistics, etc. Menshen hardware primitives

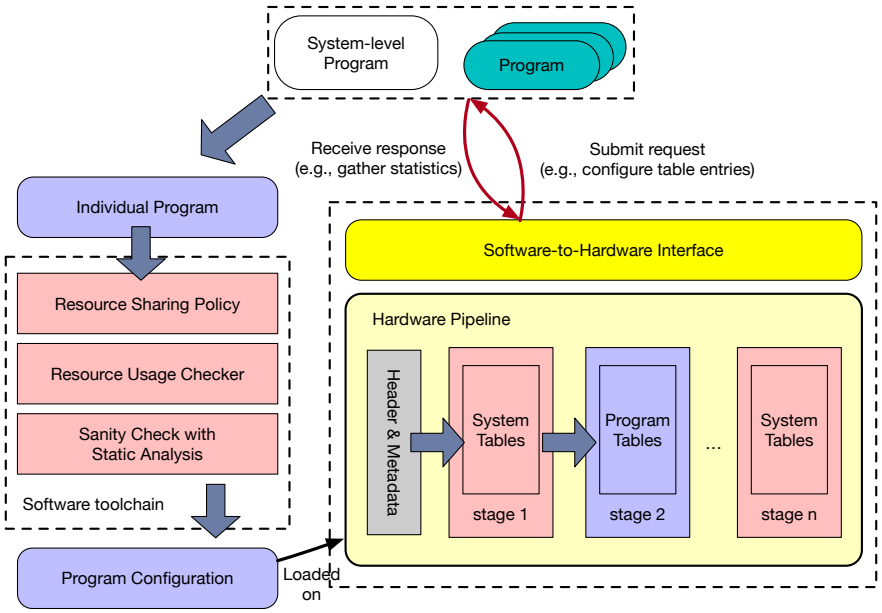


Figure 3.2: Overview of Menshen's software toolchains and hardware primitives.

(§3.3.2) primarily are in the form of small indirection tables to realize both the techniques of *space partitioning* and *overlays*.

3.3.1 Menshen Software

Menshen system-level program To hide information about the underlying physical infrastructure (*e.g.,* topology) from tenant programs in a virtualized environment, programs in Menshen can use virtual IP addresses to operate in a shared environment [92]. Here, virtual IP addresses are local and scoped to programs belonging to a particular tenant, regardless of which physical device these programs are on. To support virtual IPs and provide basic services to other programs, Menshen contains a system-level program written in P4-16 that provides common OS-like functionality, *e.g.,* converting virtual IPs to physical IPs, multicast, and looking up physical IPs to find output ports. The system-level program has 3 benefits: (1) it avoids duplication among different programs re-implementing common functions, improving the resource efficiency of the pipeline; (2) it hides underlying physical details (*e.g.,* topology) from each program so that one tenant's programs on

different network devices can form a virtual network [92]; (3) it provides common and useful real-time statistics (*e.g.,* link utilization, queue length, etc.) that can be further used by the packet processing within programs.

The bottom right corner of Figure 3.2 shows how the system-level program is laid out relative to the other user programs. Packets entering the Menshen pipeline are first processed by the system-level program before being handed off to their respective program for program-specific processing. After program-specific processing, these packets enter the system program for a second time before exiting the pipeline. The first time they enter the system-level program, packets can read and update system-level state (*e.g.,* link utilization, packet counters, queue measurements), whereas the second time they enter the system-level program, program-specific packet header fields (*e.g.,* virtual IP address) can be read by the system-level program to determine device-specific information (*e.g.,* output port). In both halves, there is a narrow interface by which programs communicate with the system-level program. This split structure of the system-level program arises directly from the feed-forward nature of the RMT pipeline, where packets typically only flow forward, but not backward. Hence, packets pick up information from the system-level program in the first stage and pass information to the system-level program in the last stage. The non-system programs are sandwiched between these two halves.

The Menshen compiler. Packet-processing pipelines (*e.g.,* RMT [73]) are structured as feed-forward pipelines of programmable units, each of which has limited processing capabilities. This design ensures the *all-or-nothing* property: once a program has been compiled and loaded it can run at up to line rate, while programs that can not run at line rate cannot be compiled. Menshen's compiler follows the same design, and only admits programs that meet line-rate requirements.

The compiler reuses the frontend and midend of the open-source P4-16 reference compiler [39] and creates a new backend similar to BMv2 [5]. This backend has a parser, a single processing pipeline, and a deparser. The compiler takes a program's P4-16 program as input and conducts all the resource usage and static checks described below. Then, for the parser and deparser, it translates

the parser defined in the program into configuration entries for the parser and deparser tables. For the packet-processing pipeline, which consists of match-action tables, it transforms the key in a table to a configuration in the key extractor table, and actions to action table entries according to the opcodes. The compiler also performs dependency checking [73, 101] to guarantee that all ALU actions and key matches are placed in the proper stage, respecting table dependencies.

The Menshen compiler can be extended to support the same packet flowing through different P4 programs belonging to one tenant. The compiler can take multiple P4 programs as input, assign them the same program ID, and allocate them to non-overlapping pipeline stages—similar to how we lay out user and system programs in different stages as in Figure 3.2.

The Menshen resource checker. The Menshen resource checker ensures that each program's resource allocation complies with an operator-specified resource sharing policy (*e.g.,* dominant resource sharing (DRF) [88], or a utility-based [95] policy). In our current design, we check allocations statically because reassigning resources from one program to another disrupts processing for both programs. Instead, we rely on admission control and do not load a program whose resource requirements cannot be met. We leave the question of what is an appropriate resource allocation policy to future work.

The Menshen static checker. To ensure isolation, Menshen's static checker analyzes 3 properties of the program's P4 source code. First, it checks that programs do not modify hardware-related statistics (*e.g.,* link utilization) provided by the system-level program to all programs. Second, programs can not modify their VID. This is because a program can be spread across multiple programmable devices [99, 87], and changes to VIDs by program *A* on a device can unintentionally affect a program *B* on a downstream device, where *B*'s real VID happens to be the same as *A*'s modified VID. Third, programs must not recirculate packets and their routing tables should be loop-free.[1] This is because all programs share the same ingress pipeline bandwidth. Recirculating packets or looping them back through multiple devices will degrade the ability of other programs

[1] We check loop freedom in the control plane.

to process packets by consuming excessive ingress bandwidth.

The software-hardware interface. The Menshen software-to-hardware interface works similarly to P4Runtime [40] to support interactions (*e.g.,* modifying match-action entries, fetching hardware statistics, etc.) between the Menshen software and the Menshen hardware. However, in addition to P4Runtime's functions, Menshen's software-hardware interface can also be used to reconfigure different hardware resources (we detail the types of Menshen resources in §3.4) in Menshen to reprogram them when a program is added or updated. This allows us to dynamically reconfigure portions of Menshen as program logic changes.

3.3.2 Menshen Hardware

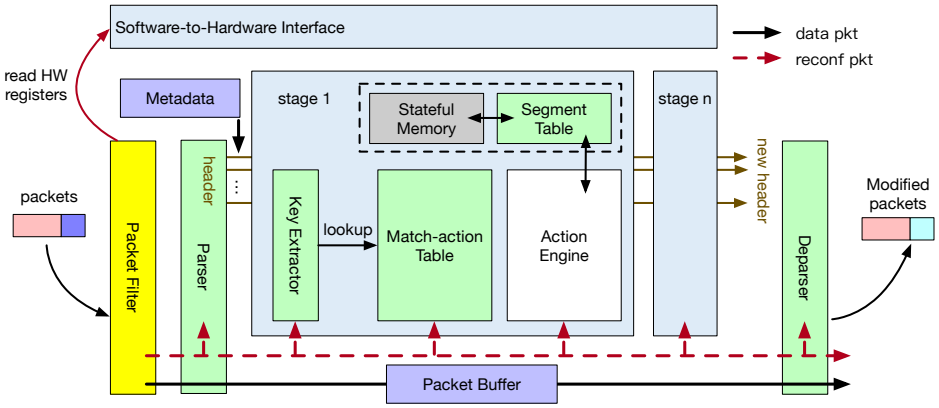


Figure 3.3: Menshen builds atop an RMT [73] pipeline. Specifically, Menshen introduces Yellow components and modifies Green ones.

As depicted in Figure 3.3, Menshen's hardware primitives are pervasively throughout the classical RMT [73] pipeline. We first describe the overall Menshen hardware design and then summarize the new isolation primitives added by Menshen.

Menshen expects that a data packet's header carries information identifying what program should process the packet. Currently, in our prototype, this is the VLAN ID (VID) header, which we assume is set by the vSwitch [92], but other fields, *e.g.,* VxLAN ID, can also be used instead. Packets entering Menshen are first handled by a packet filter that discards packets without a VLAN

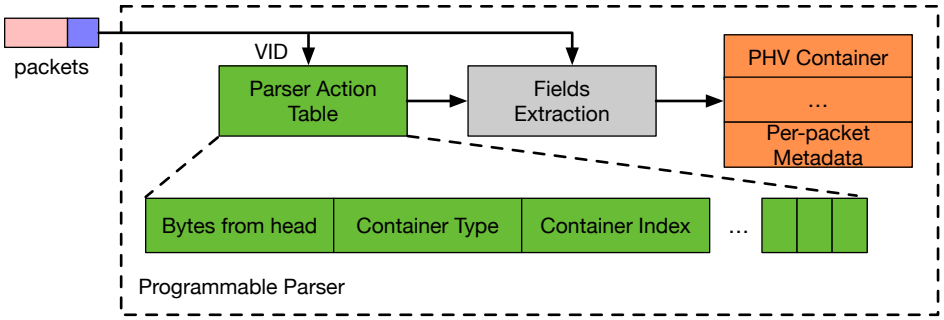


Figure 3.4: Menshen programmable parser.

ID.[2] Next, a parser extracts the VLAN ID from the packet and applies program-specific parsing to extract program-specific headers from the TCP/UDP payload. The parser then pushes these parsed packet headers into PHV containers that travel through the pipeline of match-action stages.

Each stage forms keys out of headers, looks up the keys in a match-action table, and performs actions. At the start of each stage, a key extractor in the stage forms a key by combining together the headers in a program-specific manner. The keys are then concatenated with the program ID and looked up in a match-action table, whose space is partitioned across different programs. If the key matches against a match-action pair in the table, the lookup result is used to index an action table.

Similar to the match-action table, the action table is also partitioned across programs. Each action in the table identifies opcodes, operands, and immediate constants for a very-large instruction word (VLIW), controlling many parallel arithmetic and logic units (ALUs). The VLIW instruction consumes the current PHV to produce a new PHV as input for the next stage. The table's action can modify the persistent pipeline state, stored in stateful memory. Stateful memory is indexed by a physical address that is computed from a local address, obtained from a program's packets. This computation is done by a segment table, which stores the offset and range of each program's slice of stateful memory. We now detail the main components of our design.

Parser. The Menshen parser is driven by a table lookup process similar to the RMT parser [73, 89]. Specifically, whenever a new packet comes in, the program ID is extracted from its VLAN ID

[2]The filter can be configured to send control packets without VLAN tags, *e.g.,* BFD packets [7], to the control plane or system-level program (§3.3.1).

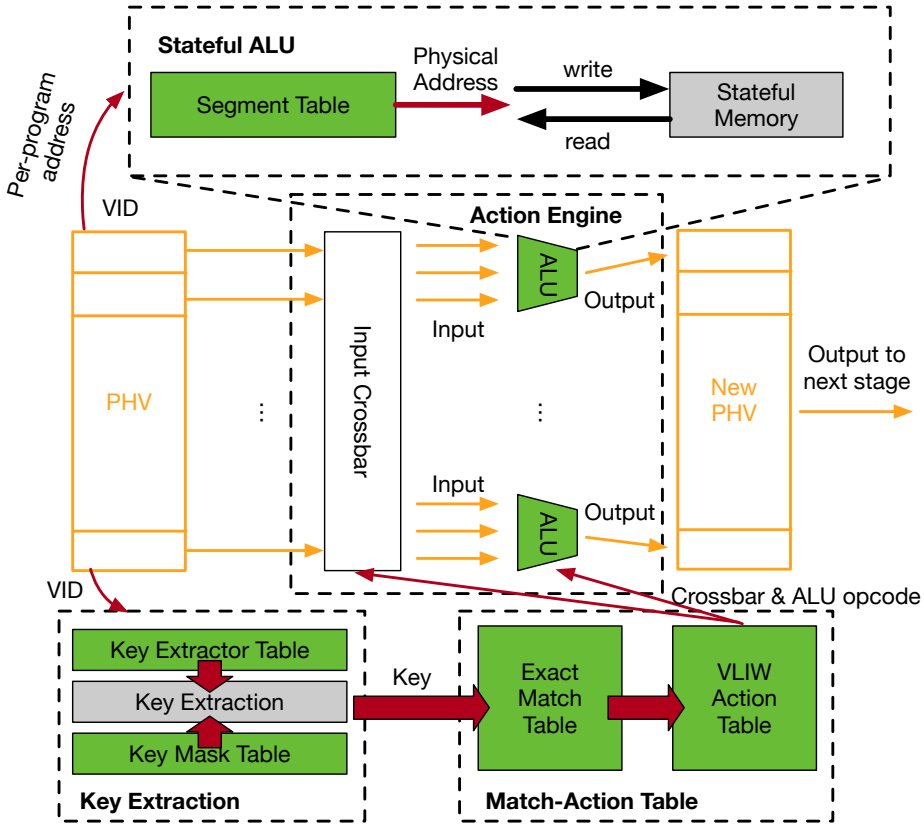


Figure 3.5: Menshen processing stage.

prior to parsing the rest of the packet. This program ID is then used as an index into the table that determines how to parse the rest of the packet (Figure 3.4). Each table entry corresponds to multiple parsing actions for a program—one action per extracted PHV container. Each parsing action specifies (1) *bytes from head*, indicating where in the packet the parser should extract a particular header; (2) *container type* (*e.g.,* 4-byte container, etc.), indicating how many bytes we should extract; (3) *container index*, indicating where in the PHV we should put the extracted header into. The parser also sets aside space in the PHV for metadata that is automatically created by the pipeline (*e.g.,* time of enqueue into switch output queues and queueing delay after dequeue) and for temporary packet headers used for computation.

Key extractor. Before a stage performs a lookup on a match-action table, a lookup key must be constructed by extracting and combining together one or more PHV containers. This key extraction

process differs between programs in the same stage, and between different stages for the same program. To implement key extraction, just like the parser, we use a key extractor table (Figure 3.5) that is indexed by a packet's program ID. Each entry in this table specifies which PHV containers to combine together to form the key. These PHV containers are then selected into the key using a multiplexer for each portion of the key. To enable variable-length key matching for different programs, the key extractor also includes a key mask table, which also uses the program ID as an index to determine how many bits to pad to a certain fixed key size before lookup.

Match table. Each stage looks up the fixed-size key constructed by the key extractor in a match table. Currently, we support only exact-match lookup. The match table is statically partitioned across programs by giving a certain number of entries to each program. To enforce isolation among different programs, the program ID is appended to the key output by the key extractor. This augmented key is what is actually looked up against the entries in the match table; each entry stores both a key and the program ID that the key belongs to. The lookup result is used as index into the VLIW action table to identify a corresponding action to execute.

Action table and action engine. Each VLIW action table entry indicates which fields from the PHV to use as ALU operands (*i.e.,* the configuration of each ALU's operand crossbar) and what opcode should be used for each ALU controlled by the VLIW instruction (*i.e.,* addition, subtraction, etc.). Each ALU outputs a value based on its operands and opcode. There is one ALU per PHV container, removing the need for a crossbar on the output because each ALU's output is directly connected to its corresponding PHV container. After a stage's ALUs have modified its PHV, the modified PHV is passed to the next stage.

Stateful memory. Menshen's action engines can also modify the persistent pipeline state on every packet. Each program is assigned its own address space, and the available stateful memory in Menshen is partitioned across programs. When a program accesses its slice of stateful memory, it supplies a per-program address that is translated into a physical address by a segment table before accessing the stateful memory. To perform this translation, Menshen stores per-program

configuration (*i.e.,* base address and range) in a segment table, which can be indexed by the packet's program ID. Menshen borrows this idea of a segment table from NetVRM's [142, 136] page table, but implements it in hardware instead of programming it in P4 atop Tofino's stateful memory like NetVRM does. This allows Menshen to avoid using scarce Tofino stateful memory to emulate a segment table. Also, by adding segment table hardware to each stage, Menshen avoids sacrificing the first stage of stateful memory for a segment table, instead reclaiming it for useful packet processing. This is unlike NetVRM, which can share stateful memory across programs only from the second stage because the first stage is used for the page table.

Deparser. The deparser performs the inverse operation of the parser. It takes PHV containers and writes them back into the appropriate byte offset in the packet header, merges the packet header with the corresponding payload in the packet buffer, and transmits the merged packet out of the pipeline. The format of the deparser table is identical to the parser table and is similarly indexed by a program ID.

Secure reconfiguration. Our threat model assumes that the Menshen hardware and software are trusted, but that data packets that enter the Menshen pipeline are untrusted. Data packets are untrusted because for a switch, they can come from physical machines outside the switch's control and, for a NIC, they can come from tenant VMs sharing the NIC. Hence, the pipeline should be reconfigured only by Menshen software, not data packets.

This is a security concern faced by existing RMT pipelines as well, even without isolation support. Commercial programmable switches solve this problem by using a separate daisy chain [11] to configure pipeline stages. This chain carries configuration commands that are picked up by the intended pipeline stage as the command passes that stage. The chain is only accessible over PCIe, which is connected to the control-plane CPU, but not by Ethernet ports, which carry outside data packets. Hence, the only way to *write* new configurations into the pipeline is through PCIe. The packet-processing pipeline is restricted to just *reading* configurations and using them to implement packet processing. Thus, the daisy chain provides secure reconfiguration by physically separating

reconfiguration and packet processing.

Menshen uses a similar approach by employing a daisy chain for reconfiguration when a program is updated. A special *reconfiguration packet* carries configuration commands for the pipeline's resources (*e.g.,* parser). Our implementation of this daisy chain varies depending on the platform. For our NetFPGA prototype, this daisy chain is connected solely to the switch CPU via PCIe, similar to current switches. For our Corundum NIC prototype, we connect the daisy chain directly to PCIe and use a *packet filter* before our parser to filter out reconfiguration packets from untrusted data packets by ensuring that reconfiguration packets have a specific UDP destination port. An ideal solution would be to use a physically separate interface, *e.g.,* USB or JTAG, for reconfiguring the Menshen pipeline on Corundum, but we found it challenging to implement such a physically separate reconfiguration interface on Corundum. In the evaluation, we show how a daisy chain permits more rapid reconfiguration than an alternative approach of using the AXI-L protocol on an FPGA.

Summary of Menshen's new primitives. The hardware primitives introduced by Menshen on top of an RMT pipeline (Figure 3.3) are the configuration tables for the parser, deparser, key extractor, key mask units and segment table. These tables provide an overlay feature to share the same unit across multiple programs. Specifically, for each unit, Menshen provides a table with a configuration entry per program, rather than one configuration for the whole unit. In addition, Menshen introduces the packet filter to ensure secure reconfiguration. Menshen also modifies match tables, by appending the program ID to the match key and the match-action entries. Finally, Menshen partitions match-action tables and stateful memory across all programs. These primitives ensure that updating one program only affects a single entry (for Menshen resources that use overlays) and only affects a subset of memory (for Menshen resources that use space partitioning), thus allowing us to update one program without disrupting others (§3.5).

ASIC feasibility of Menshen's primitives. Menshen's parser, deparser, key extractor, key mask, and segment tables are small and simple arrays indexed by the program identifier. They can be

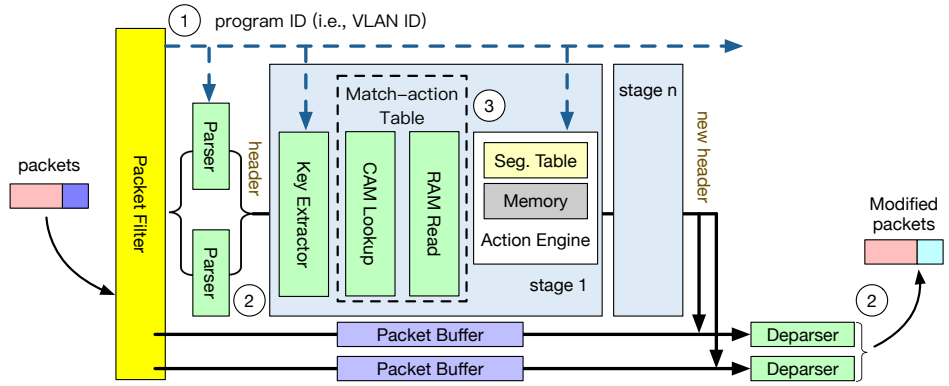


Figure 3.6: Three optimization techniques applied in Menshen. Numbered circles refer to specific techniques detailed in §3.3.3.

readily realized in SRAM that can support a memory read every clock cycle. The packet filter is a simple combinational circuit that checks if the incoming packet is destined to a specific UDP destination port. Extending the match-action tables in each stage to append a program ID to every entry amounts to modestly increasing the key width in the table. While these new primitives add some additional latency relative to RMT, *e.g.,* to go through the packet filter or reading out the per-program parser configuration, the pipelined nature of RMT means that this additional latency does not impact the packet-forwarding rate.

3.3.3 Optimizations for Menshen Hardware

As shown in Figure 3.6, we apply 3 main techniques to optimize the forwarding performance of Menshen: (1) masking RAM read latency, (2) using multiple parsers and deparsers, and (3) increasing pipeline depth. We demonstrate the effect these techniques have on Menshen when evaluating Menshen's throughput in §3.5.2.

(1) ***Masking RAM read latency.*** The design described in §3.3.2 attaches the module ID to the PHV that is sent from one element (e.g., parser, key extractor) to the next. In this design, we read the module's configuration from SRAM after the PHV arrives, thus incurring a few additional

clock cycles of latency. To optimize this, we mask SRAM access latency by splitting the module ID from the PHV and sending the module ID to the next element *ahead of time*. The PHV follows the module ID, and thus the module configuration at a stage can be read concurrently with the PHV being transmitted to that stage.

(2) ***Multiple parsers and deparsers.*** In §3.3.2's design, there is one parser, deparser, and packet buffer. The parser extracts and parses the header and puts the full packet in the packet buffer. Then the deparser takes the modified headers from the last stage, uses them to overwrite the relevant portions of the full packet in the packet buffer, and sends out the packet.

Our optimized design uses multiple parallel parsers, deparsers, and packet buffers to improve throughput. Deparsing is the most expensive operation as any position within the PHV container might be modified, and thus any part of the packet header (128 bytes in our implementation) might need to be updated. Furthermore, deparsing has to process both the packet header and the payload. Therefore, we use 4 parallel deparsers and 2 parsers. We also associate a separate packet buffer with each deparser.

On ingress, the packet filter tags each packet with a packet buffer number (0–3) in round robin order. It also round robins incoming packets to the 2 parsers. The last pipeline stage uses the packet buffer tag to determine which packet buffer's packet the last stage's modified PHV should be combined with. Each packet buffer's deparser combines the earliest packet from the packet buffer along with the last stage's most recently modified PHV for that buffer.

(3) ***Deep pipelining.*** With careful digital design, in Menshen's implementation, we can pipeline each element (e.g., match-action table) into several sub-elements to improve throughput. For example in Figure 3.6, we divide the match-action table into CAM-lookup and action-RAM-read sub-elements. In this specific example, this allows us to process a PHV every 2 clock cycles at each sub-element rather than every 4 clock cycles at the whole match-action table.

3.4 Implementation

3.4.1 Menshen Software

The Menshen compiler reuses the open-source P4-16 reference compiler [39] and implements a new backend extension in 3773 lines of C++. It takes the program written in P4-16 together with resource allocation as the inputs, and generates per-program configurations for Menshen hardware. Specifically, it (1) conducts resource usage checking to ensure every program's resource usage is below its allocated amount; (2) places the system-level program's (120 lines of P4-16) configurations in the first and last stages in the Menshen pipeline; and (3) allocates PHV containers to the fields shared between the system-level and other programs so that the other programs can be sandwiched between the two halves of the system-level program (§3.3.1). The Menshen software-to-hardware interface is written in Python. It configures Menshen hardware by converting program configurations to reconfiguration packets.

3.4.2 Menshen Hardware

To implement Menshen hardware, we first built a baseline RMT implementation for an FPGA. Menshen includes (1) a packet filter to filter out reconfiguration packets from data packets using a specific predefined UDP destination port (*i.e.,* 0xf1f2), (2) a programmable parser, (3) a programmable RMT pipeline with 5 programmable processing stages, (4) a deparser, and (5) a separate daisy-chain pipeline for reconfiguration. It also includes Menshen's primitives for isolation. We have integrated it into both the Corundum NIC [83] and the NetFPGA reference switch [144]. The Menshen code base together with the optimizations (§3.3.3) consists of 9975 lines of Verilog. Of this, 3098 and 3226 lines are for handling data bus widths of 512 bits (Corundum) and 256 bits (NetFPGA) respectively. 3651 lines are for the common blocks, *e.g.,* key extractor, etc. Below, we describe our hardware implementation in more detail. Figure 3.7 shows the formats of Menshen's

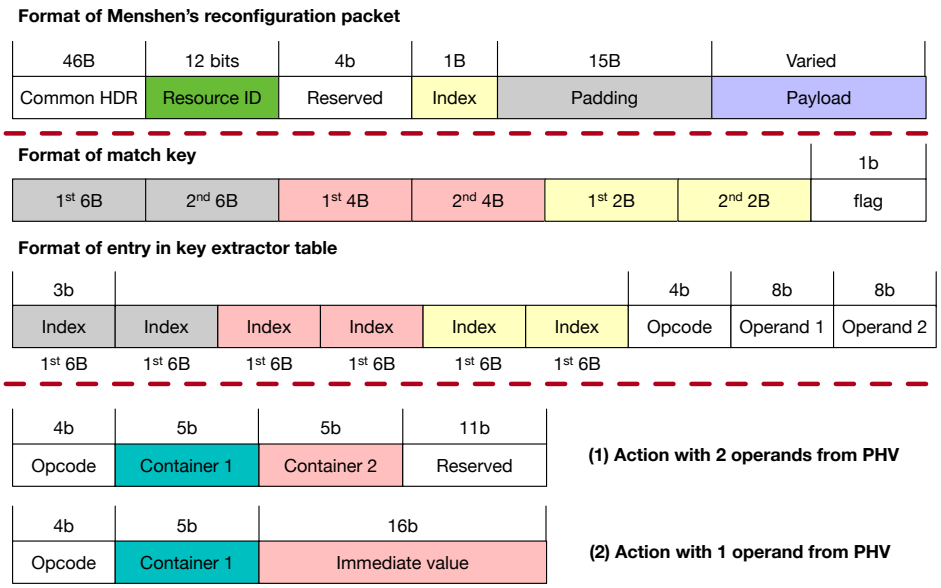


Figure 3.7: Formats of Menshen's packets and tables.

packets and tables.

PHV format. Menshen's PHV has 3 types of containers of different sizes, namely 2-byte, 4-byte and 6-byte containers. Each type has 8 containers. Also, we allocate and append an additional 32 bytes to store platform-specific metadata (*e.g.,* an indication to drop the packet, destination port, etc.), which results in a PHV length of 128 bytes in total. Thus, we have a total of $3*8+1=25$ PHV containers. To prevent any possibility of PHV contents leaking from one program to another, the PHV is zeroed out for each incoming packet.

Reconfiguration packet format. Figure 3.7 shows the format of Menshen reconfiguration packets. The reconfiguration packet is a UDP packet with the standard UDP, Ethernet, VLAN, and IP headers. Within the UDP payload, a 12-bit resource ID indicates which hardware resource within which stage should be updated (*e.g.,* key extractor table in stage 3). To reconfigure the resource, the table storing the configuration for this resource must be updated by writing the entry stored within the reconfiguration packet's payload at the location specified by the 1-byte index field in the reconfiguration packet header. The UDP destination port field determines whether the reconfiguration packet is valid or not.

Packet filter. The packet filter has 2 registers that can be accessed by the Menshen software via Xilinx's AXI-Lite protocol [61]: (1) a 4-byte reconfiguration packet counter, which monitors how many reconfiguration packets have passed through the daisy chain; (2) a 32-bit bitmap, which indicates which program is currently being updated (*e.g.,* bit 1 stands for program 1, bit 2 for program 2, etc.). During the reconfiguration of a program, via the software-to-hardware interface, the Menshen software reads the reconfiguration packet counter. It then writes the bitmap to reflect the program ID M of the program currently being updated. The bitmap is then consulted on every packet to drop data packets from M until reconfiguration completes, so that M's "in-flight" packets aren't incorrectly processed by partial configurations.

Then, the Menshen software sends all reconfiguration packets embedded with the predefined UDP destination port to the daisy chain. Finally, it polls the reconfiguration packet counter to check if reconfiguration is over and then zeroes the bitmap so that M's packets are no longer dropped. Reconfiguration packets may be dropped before they reach the RMT pipeline. This can be detected by polling the reconfiguration packet counter to see if it has been correctly incremented or not. If it hasn't been incremented correctly, then the entire reconfiguration process restarts with M's packets being dropped until reconfiguration is successful.

Programmable parser/deparser. We currently support per-program packet header parsing in the first 128 bytes of the packet. These 128 bytes also include the headers common to all programs (*e.g.,* Ethernet, VLAN, IP, and UDP). We design the parser action for each parsed PHV container as a 16-bit action. The first 3 bits are reserved. The next 7 bits indicate the starting extraction position in bytes from byte 0. These 7 bits can cover the whole 128-byte length. Then, the next 2 bits and 3 bits indicate the container type (2, 4, or 6 byte) and number (0–7) respectively. The last bit is the validity bit. For each program, we allocate 10 such parser actions (*i.e.,* to parse out at most 10 containers), resulting in a 160-bit-wide entry for the parser action table.

We note that we only parse out fields of a packet into PHV containers, if those fields are actually used as part of either keys or actions in match-action tables. Before packets are sent out, the

deparser pulls out the full packet (including the payload) from the packet buffer and only updates the portions of the packet that were actually modified by table actions. This approach allows us to reduce the number of PHV containers to 25 because packet fields that are never modified or looked up by the Menshen pipeline need not travel along with the PHV.

Key extractor. The key for lookup in the match-action table is formed by concatenating together up to 2 PHV containers each of the 2-byte, 4-byte, and 6-byte container types. Hence the key can be up to 24 bytes and 6 containers long. Since there are 8 containers per type, the key extraction table entry for each program in each stage uses $log_2(8) * 6 = 18$ bits to determine which container to use for the 6 key locations. Additionally, the key extractor is also used to support conditional execution of actions based on the truth value of a predicate of the form $A\ OP\ B$, where A and B are packet fields and OP is a comparison operator. For this purpose, each key extractor table entry also specifies the 2 operands for the comparison operation and the comparison opcode. The opcode is a 4-bit number, while the operands are 8 bits each. The operands can either be an immediate value or refer to one of the PHV containers. The result of the predicate evaluation adds one bit to the original 24 byte key, bringing the total key length to $24 * 8 + 1 = 193$ bits. Because not all keys need to be 193 bits long, we use a 193-bit-wide mask table. Each entry in this table denotes the validity of each of the 193 key bits for each program in each stage. This is somewhat wasteful and can be improved by storing validity information within the key extractor table itself.

Exact match table. To implement the exact match table, we leverage the Xilinx CAM block [64]. This CAM matches the key from the key extractor program against the entries within the CAM. As discussed in §3.3.2, to ensure isolation between different programs, we append the program ID (*i.e.,* VLAN ID) to each entry, which means that the CAM has a width of $193 + 12 = 205$ bits. The lookup result from the CAM is used to index the VLIW action table. The action is designed in a 25-bit format per ALU/container (Figure 3.7). As we have $24 + 1 = 25$ PHV containers, the width of the VLIW action table is $25 * 25 = 625$ bits. The Xilinx CAM block simplifies the implementation of an exact-match table and can also easily support ternary matches if needed.

Operation	Description
`add/sub`	Add/subtract between containers
`addi/subi`	Add/subtract an immediate to/from container
`set`	Set a container to an immediate value
`load`	Load a value from stateful memory
`store`	Store a value to stateful memory
`loadd`	Load value from stateful memory, add 1, and store back
`port`	Set destination port
`discard`	Discard packet

Table 3.2: Supported operations in Menshen's ALU.

Action engine. The crossbar and ALUs in the action engine use the VLIW actions to generate inputs for each ALU and carry out per-ALU operations. ALUs support simple arithmetic, stateful memory operations (*e.g.,* loads and stores), and platform-specific operations (*e.g.,* discard packets) (Table 3.2). The formats of these actions are shown in Figure 3.7. Additionally, in stateful ALU processing, each entry in the segment table is a 2-byte number, where the first byte and second byte indicate memory offset and range, respectively.

Menshen primitives. Menshen's isolation primitives (*e.g.,* key-extractor and segment tables) are simple arrays implemented using the Xilinx Block RAM [63] feature. All Menshen's hardware resources are detailed in Table 3.3.

3.4.3 Corundum and NetFPGA Integrations

We have integrated Menshen into 2 FPGA platforms: one for the NetFPGA platform that captures the hardware architecture of a switch [144], and another for the Corundum platform that captures the hardware architecture of a NIC [83]. Menshen's integration on Corundum [83] is based on a 512-bit AXI-S [62] data width and runs at 250 MHz. Although Menshen's pipeline can be integrated into both the sending and receiving path, in our current implementation, we have integrated Menshen into only Corundum's sending path, *i.e.,* PCIe input to Ethernet output.

Hardware Resource	Description
Packet Filter	A 32-bit bitmap, and a 4-byte reconfiguration packet counter
PHV	2-byte, 4-byte, 6-byte containers, each type has 8 containers a 32-byte container for platform-specific metadata
Parsing action	16 bits wide
Parser and deparser table	10 parsing actions, 160 bits wide, 32 entries deep
Key extractor table	38 bits wide, 32 entries deep
Key mask table	193 bits wide, 32 entries deep
Exact match table	205 bits wide, 16 entries deep
ALU Action	25 bits wide
VLIW action table	25 ALU actions, 625 bits wide, 16 entries deep
Segment table	16 bits wide, 32 entries deep
Stages	5
Program ID	12 bits

Table 3.3: Hardware resources in Menshen

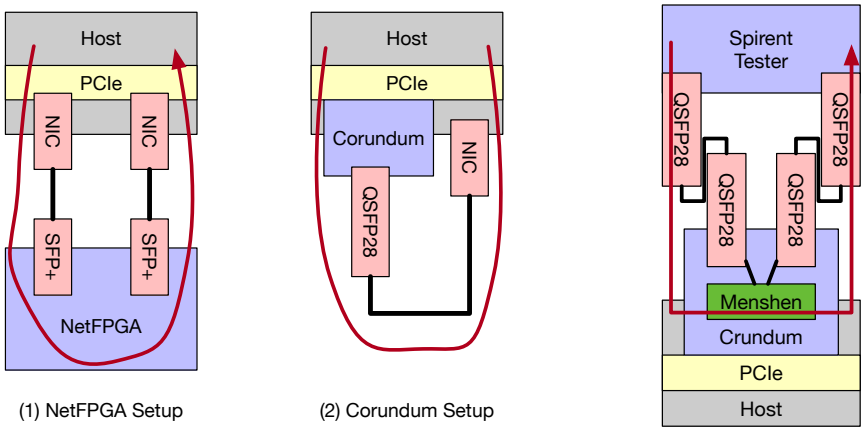
Menshen on NetFPGA [144] uses a 256-bit AXI-S [62] data width and runs at 156.25 MHz.

On the Corundum NIC platform, we insert a 1-bit discard flag, while on the NetFPGA switch platform, we insert a 1-bit discard flag and 128-bit platform-specific metadata, *i.e.,* source port, destination port and packet length, into the PHV's metadata field. A 4-bit one-hot encoded tag indicates the packet buffer (§3.3.3). The table depth in Menshen's parser, key extractor, key mask, page, and deparser tables affects the maximum number of programs we can support and is currently 32. The depth of CAM and VLIW action table directly influences the amount of match-action entries and VLIW actions that can be allocated to all programs. Due to the open technical challenge of implementing CAMs on FPGAs efficiently [120, 98], we set their depth to 16 in each stage. While 16 is a small depth, the depth can be improved by using a hash table, rather than a CAM, for exact matching, *e.g.,* cuckoo hashing [117].

Program	Description
CALC [41]	return value based on parsed opcode and operands
Firewall [41]	stateless firewall that blocks certain traffic
Load Balancing [41]	steer traffic based on 4-tuple header info
QoS [41]	set QoS based on traffic type
Source Routing [41]	route packets based on parsed header info
NetCache [100]	in-network key-value store
NetChain [99]	in-network sequencer
Multicast [41]	multicast based on destination IP address

Table 3.4: Evaluated use cases.

3.5 Evaluation



(a) Correctness setup. **(b)** Performance test setup.

Figure 3.8: Testbed setup. Red arrow shows packet flow.

In §3.5.1, we show that Menshen can meet our requirements (§3.1): it can be rapidly reconfigured, is lightweight, provides behavior isolation, and is disruption-free. Menshen achieves performance isolation by (1) assuming packets exceed a minimum size (to guarantee line rate) and (2) forbidding recirculation. If either is violated, hardware rate limiters can be used to limit each program's packet/bit rate. It achieves resource isolation by ensuring that a table entry for a resource (*e.g.,* parser) is allotted to at most one program. In §3.5.2, we evaluate the current performance of

Menshen in terms of throughput and latency.

Experimental setup. To demonstrate Menshen's ability to provide multi-program support, we picked 6 tutorial P4 programs [41], as detailed in Table 3.4, together with simplified versions of NetCache [100] and NetChain [99].[3] The system-level program provides basic forwarding and routing, with multicast logic integrated into it. Menshen's parameters are detailed in §3.4 and summarized in Table 3.3 in the §3.4.2.

Testbed. Our experimental setup is depicted in Figure 3.8. We evaluate Menshen based on our Corundum and NetFPGA integrations as described in §3.4.3. For the switch platform experiments on NetFPGA, we use a single quad-port NetFPGA SUME board [33], where two ports are connected to a machine equipped with an Intel Xeon E5645 CPU clocked at 2.40 GHz and a dual-port Intel XXV710 10/25GbE NIC. For the NIC platform experiments on Corundum, we use a single Xilinx Alveo U250 board [3], where one port is with Menshen for the transmitting path and this port is connected to a 100 GbE NIC as the receiving path. Both setups are used to check Menshen's correctness (§3.5.1). For NetFPGA performance tests (§3.5.2), we use the host as a packet generator. For Corundum performance tests (§3.5.2), we internally connect its receiving and transmitting path, and use the Spirent tester [49] to generate traffic.

3.5.1 Does Menshen meet its requirements?

Menshen can be rapidly reconfigured. Reconfiguration time includes both the software's compilation time (Figure 3.9) and the hardware's configuration time (Figure 3.10); we evaluate each separately. When a program is compiled, the compiler needs to generate both configuration bits for various hardware resources as well as match-action entries for the tables the program looks up. These match-action entries can and will be overwritten by the control plane, but we need to start out with a new set of match-action entries for a program to ensure no information leaks from a previous program.

[3] Our versions of NetChain and NetCache do not include some features such as tagging hotkeys.

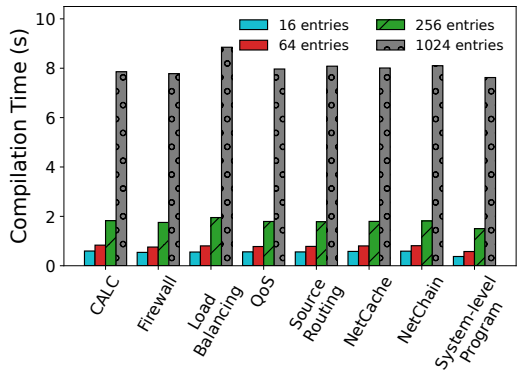


Figure 3.9: Compilation time of Menshen.

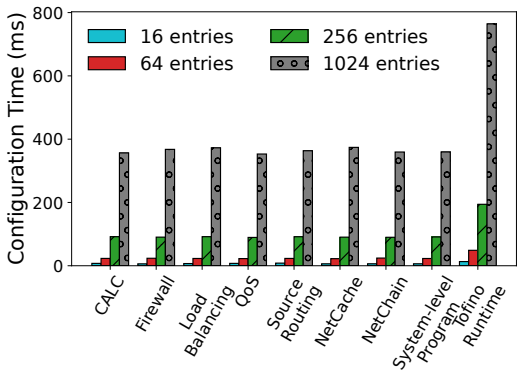


Figure 3.10: Configuration time of Menshen.

Hence, every time a program is compiled, the compiler also generates match-action entries. Within an exact match table, these entries must be different from each other to prevent multiple lookup results. As a result shown in Figure 3.9, Menshen's compilation time increases with the number of match-action entries in the program. To contextualize this, Menshen's compile times (few seconds) compare favorably to compile times for Tofino (~10 seconds for our use cases) and FPGA synthesis times (10s of minutes). We note that this is an imperfect comparison: our compiler performs fewer optimizations than either the Tofino or FPGA compilers and our targets are simpler. That said, compilation can happen offline, and hence it is not as time-sensitive compared to run-time reconfiguration which is more important as new configurations should take effect as quickly as possible.

To measure time taken for Menshen's configuration post compilation, we vary the number of entries the Menshen software has to write into the pipeline.[4] Also, as a comparison, we evaluate the cost of the Tofino run-time APIs from Tofino SDE 9.0.0 to insert match-action table entries for the CALC program. From Figure 3.10, we observe that the time spent in configuration of the hardware via Menshen's software-to-hardware interface is similar to Tofino's run-time APIs.

Daisy-Chain vs. Fully-AXI-L-Based Configuration. Additionally, as discussed in §3.4.2, Men-

[4]Since the Menshen hardware can't currently support so many entries (§3.4.2), we overwrite previously written entries to measure configuration time.

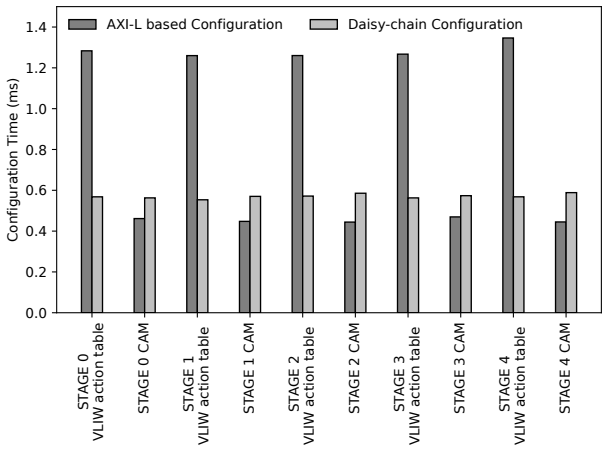


Figure 3.11: Configuration time comparison for AXI-L based (estimated) and Menshen's daisy-chain configuration (measured).

shen uses a daisy chain pipeline to configure the Menshen pipeline and uses the AXI-L [61] protocol for safety alone, *i.e.,* to read the reconfiguration packet counter and update the bitmap during reconfiguration. Before using this daisy-chain approach, we considered a different approach based fully on the AXI-L protocol. In this approach, all configuration settings on the FPGA would be set using the AXI-L protocol via PCIe from the host instead of passing a reconfiguration packet through a daisy chain pipeline. We elected to use the daisy-chain approach instead for 2 reasons described below.

First, as one AXI-L write in Corundum can only support a 32-bit data length, we have to write $\lceil 625/32 \rceil = 20$ and $\lceil 205/32 \rceil = 7$ times for configuring one entry in the VLIW action table and CAM respectively. For our test modules, we estimate AXI-L reconfiguration time based on the write time of a single AXI-L write. As shown in Figure 3.11, Menshen's daisy-chain configuration is much faster than the AXI-L based method, especially for longer entries (*i.e.,* VLIW action table). These benefits are likely to be more pronounced on a larger implementation of Menshen because the entries (both for VLIW action table and CAM) will be even longer in that case. Second, the daisy-chain approach is more similar in style to how programmable switch ASICs are configured today, hence, it is preferable for an eventual ASIC implementation of Menshen.

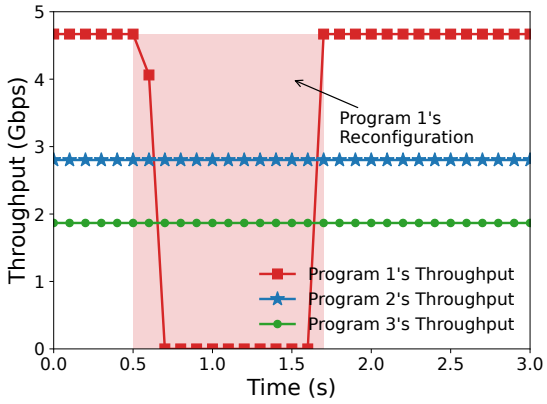


Figure 3.12: Disruption-free reconfiguration measured every 0.1 s.

Menshen can reconfigure without disruption. To show Menshen can support disruption-free reconfiguration, we launch three CALC programs with fixed input packet rates, *i.e.,* 5:3:2 ratio on a single link for the program 1, 2 and 3, respectively. We use netmap-based tcprelay to generate total traffic of 9.3 Gbit/s on a 10 Gbit/s link. 0.5 seconds in, we start to reconfigure the first program to see if the packet processing of other programs has stalled or not. In Figure 3.12 we show the throughput achieved by each of the three programs when reconfiguring program 1. We can observe that model 2 and 3 see no impact on their throughput. This demonstrates that Menshen provides performance isolation, and that it is feasible for a tenant to reconfigure their program without impacting other tenants. By contrast, updating a program on Tofino (§3.6) requires resetting the entire switch pipeline. Even with Tofino's Fast Refresh [25], this leads to a 50 ms disruption of all servers (and their VMs) whose traffic is routed through the switch. This disruption of tens of milliseconds can be significant in public cloud environments, and in many cases renders dynamic reconfiguration infeasible.

Menshen is lightweight. We list Menshen's resource usage of logic and memory (*i.e.,* LUTs and Block RAMs), including absolute numbers and fractions, in Table 3.5. For comparison, we also list the resource usage of the NetFPGA reference switch and the Corundum NIC. We believe that the additional hardware footprint of Menshen is acceptable for the programmability

Hardware Implementation	Slice LUTs	Block RAMs
NetFPGA reference switch	42325 (9.77%)	245.5 (16.7%)
RMT on NetFPGA	200573 (46.3%)	641 (43.6%)
Menshen on NetFPGA	200733 (46.34%)	641 (43.6%)
Corundum	61463 (3.56%)	349 (12.98%)
RMT on Corundum	235686 (13.63%)	316 (11.75%)
Menshen on Corundum	235903 (13.65%)	316 (11.75%)

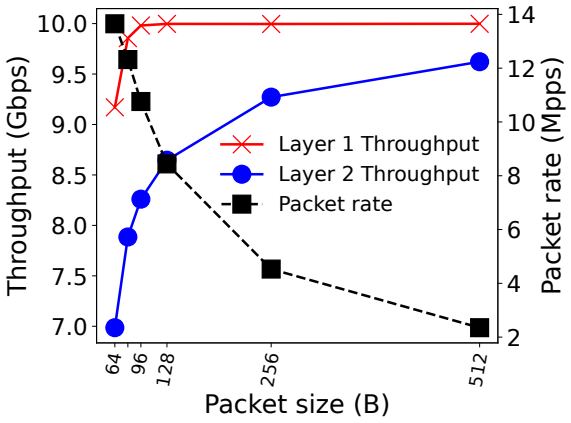
Table 3.5: Resources used by 5-stage Menshen pipeline, on NetFPGA SUME and AU250 boards, compared with reference switch, Corundum NIC, and RMT.

and isolation mechanisms it provides relative to the base platforms. The reason that Menshen uses more LUTs than Block RAMs is that Menshen leverages the Shift Register Lookup (SRL)-based implementation of Xilinx's CAM IP [64]. We also compared with an RMT design, where we modified Menshen's hardware to support only one program. Relative to RMT, Menshen incurs an extra 0.65% (NetFPGA) and 0.15% (Corundum) in LUTs usage.

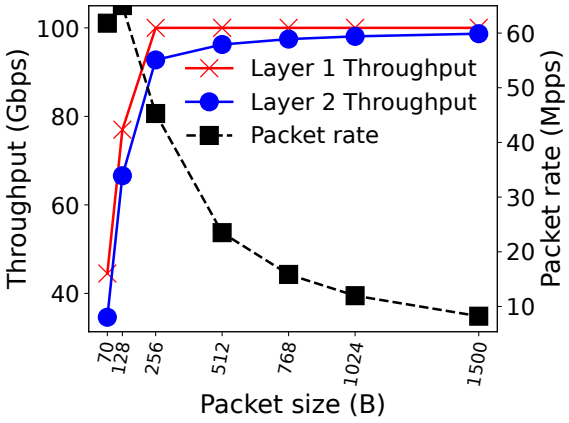
Menshen provides behavior isolation. Next, we spot check that Menshen can correctly isolate programs, *i.e.,* every running program can concurrently execute its desired functionality. For this, we ran the CALC, Firewall, and NetCache program simultaneously on the Menshen pipeline. We generate data packets of different VIDs, which indicate which of these 3 programs they belong to, and input them to the Menshen FPGA prototype on both platforms. By examining the output packets at the end of Menshen's pipeline, we checked that Menshen had correctly isolated the programs, *i.e.,* each program behaved as it would have had it run by itself. We repeated the same experiment by running the Load Balancing, Source Routing, and NetChain programs simultaneously; we observed correct behavior isolation here too.

3.5.2 Menshen Performance

How many programs can be packed? In our current prototype on both Corundum and NetFPGA, we can support at most 32 programs because each isolation primitive (*e.g.,* key extractor table)

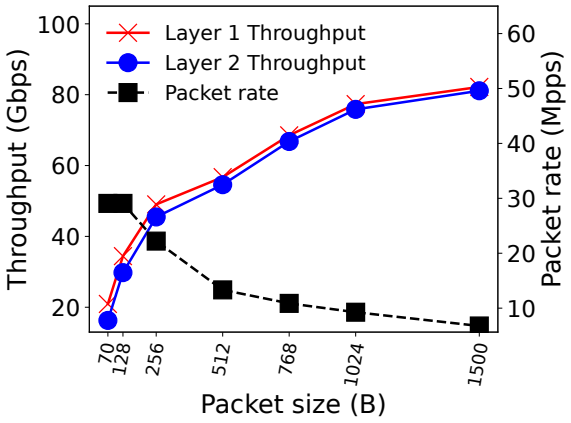


(a) Optimized NetFPGA.

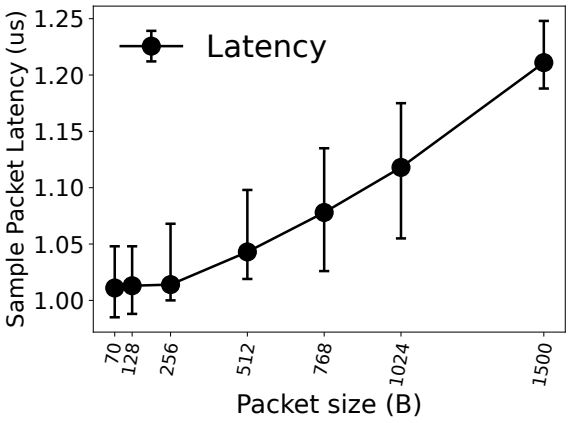


(b) Optimized Corundum.

currently has 32 entries. In practice, the number of programs could be less than 32 if programs need to share a more bottlenecked hardware resource. For instance, if each program wants a match-action entry in every pipeline stage, the maximum number of programs is at most 16 because there are only 16 match-action entries in each stage in our current prototype. However, the numbers above are entirely a function of how much hardware one is willing to pay in exchange for multitenancy support. If we can afford to expend additional resources on an FPGA or extra area on an ASIC, we can correspondingly support a larger number of programs.



(a) Unoptimized Corundum.



(b) Optimized Corundum latency.

Figure 3.14: Results for performance benchmarks.

Latency. In our current implementation, the number of clock cycles needed to process a packet in

the pipeline depends on packet size. This is because the number of cycles to process both the header and the payload depend on the header and payload length. For instance, for a minimum packet size of 64 bytes, Menshen's pipeline introduces 79 and 106 cycles of processing for NetFPGA and Corundum, resulting in $79 * \frac{1000}{156.25} = 505.6$ ns and $106 * \frac{1000}{250} = 424$ ns latency, respectively. For the max. packet size of 1500 bytes, Menshen incurs 146 and 112 cycles for NetFPGA and Corundum, resulting in $150 * \frac{1000}{156.25} = 960$ ns and $129 * \frac{1000}{250} = 516$ ns latency.

Throughput. For NetFPGA, we used MoonGen [80] to generate packets with different sizes. Figure 3.13(a) shows that Menshen achieves a rate of 10 Gbit/s after a packet size of 96 bytes. This is the maximum supported by our MoonGen setup because we have a single 10G NIC. For Corundum, we internally connected Corundum's receiving and transmitting path. Rather than using a host-based packet generator through PCIe, we used Spirent FX3-100GO-T2 tester to test Menshen's throughput. The MTU size is set to 1500 bytes. As shown in Figure 3.13(b) and Figure 3.14(a), optimized Menshen on Corundum achieves 100 Gbit/s at 256 bytes, while unoptimized Menshen can only achieve 80 Gbit/s at MTU-size packets. Also, we sample packets to evaluate the packet latency of optimized Menshen on Corundum with full rate. As depicted in Figure 3.14(b), at full rate, it incurs about 1.2 µs latency.

ASIC feasibility. With the same parameter settings in §3.5, we use the Synopsys DC synthesis tool [52] and FreePDK45nm technology library [17] to assess the ASIC feasibility of the Menshen pipeline.[5] At 1 GHz frequency, when compared with an RMT design, where we modified Menshen to support only one program, Menshen incurs 18.5%, 7%, 20.9% additional chip area for the parser, deparser and one stage, respectively. For a 5-stage pipeline along with the packet filter, parser, deparser and packet buffers, Menshen (10.81 mm^2) incurs 11.4% additional chip area compared with RMT (9.71 mm^2).

Considering that memory (*i.e.,* lookup tables) and packet processing logic only costs at most 50%

[5]Since we can not have access to the source code of Xilinx IPs (*e.g.,* DMA, Ether+PHY, etc.), we solely run synthesis on Menshen's Verilog codebase.

in switch chip area [42, page 36], Menshen's chip area overhead is moderate ($11.4\% * 50\% = 5.7\%$), which is conservative since the number of entries in our match-action table is only 16 (§3.4.2). With a much larger number of entries in lookup tables—which is the common block between Menshen and RMT—Menshen's additional chip area will be negligible.

3.6 Related Work

Multi-core architecture solutions. To support isolation on programmable network devices based on multicores [50, 27, 29], FairNIC [91] partitions cores, caches, and memory across tenants and shares bandwidth across tenants through Deficit Weighted Round Robin (DWRR) scheduling. iPipe [114] uses a hybrid DRR+FCFS scheduler to share SmartNIC and host processors between different programs. Menshen uses space partitioning as well to allocate different resources to different programs. However, RMT's spatial/dataflow architecture differs considerably from the Von Neumann architectures for multi-core network processors targeted by FairNIC and iPipe. An RMT architecture can not support a runtime system similar to the ones used by iPipe and FairNIC.

FPGA-based solutions. Several FPGA platforms exist for programmable packet processing. These platforms can be broadly categorized into (1) direct programming of FPGAs [106, 82, 30, 96, 127, 131, 133] and (2) higher-level abstractions built on top of FPGAs [120, 74, 67, 81].

Systems (*e.g.,* VirtP4 [127], MTPSA [131]) based on direct FPGA programming typically implement packet-processing logic in a hardware-description language (HDL) or using a high-level language like P4 [133, 96] or C [66, 106] that is translated into HDL. The HDL program is fed to an FPGA synthesis tool to produce a bitstream, which is written into the FPGA. This approach requires combining the programs of different programs into a single Verilog program, which can then be fed to the synthesis tool. Thus, changing one program disrupts other programs, violating our requirement of no disruption, similar to other compile-time solutions (*e.g.,* P4Visor [139], ShadowP4 [140]).

The disruption can be addressed through partial reconfiguration (PR) [105, 6.2], where the grid of computing elements (called look-up tables or LUTs) on an FPGA are divided into regions. Each region is assigned to a different Verilog program, and the FPGA bitstream in each region can be synthesized independent of the other regions. PR can be used for isolation by assigning each packet-processing program to a separate PR region. Menshen's space partitioning to allocate RMT resources across programs is effectively a form of partial reconfiguration, but specialized to the context of RMT. However, space sharing alone isn't sufficient for isolation on RMT and we additionally need overlays to share some RMT resources across multiple programs.

FlowBlaze [120], SwitchBlade [67], and hXDP [74] expose a restricted higher-level abstraction like RMT or eBPF on top of an FPGA. FlowBlaze and hXDP do not provide support for isolation. SwitchBlade does, but its higher-level abstraction is much less flexible than the RMT abstraction in Menshen. NICA [81] targets an FPGA NIC and is designed to share one pre-programmed offloading engine across many programs, while Menshen also targets ASIC pipelines and supports reprogramming individual programs without disrupting others. This higher-level abstraction is reconfigured by a compiler every time a programmer updates the program, instead of synthesizing a new FPGA bitstream each time, which is more time consuming.

Our prototype implementation of Menshen is effectively a higher-level RMT model on top of an FPGA with support for inter-program isolation. However, we stress that our FPGA prototype is for ease of engineering; an eventual implementation of Menshen will likely use an ASIC.

Tofino [22]. Tofino is a commercial switch ASIC that uses multiple parallel RMT pipelines. However, Tofino currently does not support multiple programs/P4 programs within a single pipeline. The current Tofino compiler requires a single P4 program per pipeline. Multiple P4 programs can be merged into a single program per pipeline and then fed into the Tofino compiler (Wang et al. [136] and μP4 [129]). However, both approaches still disrupt all tenants every time a single tenant in any pipeline is updated. This is because despite supporting an independent program per pipeline, updating any of these programs requires a reset of the entire Tofino switch [25].

Emulation-based solutions. Hyper4 [94] and HyperV [138] propose to emulate multiple P4 programs/programs using a single hypervisor P4 program, which can be configured at run time by the control plane, thus supporting disruption-free reconfiguration. However, we found that it was very challenging to design a sufficiently "universal" hypervisor program on a commercial RMT switch like Tofino.

As one example, the hypervisor program needs to support performing a bit-shift by an amount determined by a packet field, where the packet field is specified by the control plane. However, a high-speed chip like Tofino has several restrictions on bit-shifts and other computations for performance, *e.g.,* on Tofino, the shift width and field to shift must be supplied at compile time, not at run time by the control plane.

PANIC [112] and FlexCore [137]. PANIC and FlexCore [137] are programmable multi-tenant NIC and switch designs, respectively. They both suffer from scalability issues because they need to build a large crossbar with long wires interconnecting all engines to each other, which requires careful physical design [75, Appendix C]. Menshen's RMT pipeline is easier to scale as its wires are shorter: they only connect adjacent pipeline stages [73, 2.1].

3.7 Summary

To sum up, this chapter presents Menshen, a system for addressing the issue of accessibility in terms of isolating co-resident packet-processing modules on pipelines similar to RMT. Menshen comprises both software toolchains and hardware primitives: (1) Menshen software is mainly used for allocating hardware resources and guaranteeing no malicious operations are done by each user-submitted program; (2) Menshen hardware builds on the idea of space partitioning and overlays, and is comprised of a set of simple hardware primitives that are inserted at different points in an RMT pipeline. These primitives are straightforward to realize in both ASICs and FPGAs. Menshen thus demonstrates that providing inter-module isolation in programmable pipelines is practical.

4 | QingNiao

This chapter introduces QingNiao, a system that addresses the other aspect, *i.e.,* general L7 processing. Over the last decade, L7 processing has been increasingly moved from the application itself into the software proxies (*e.g.,* Envoy [14]). Specifically, QingNiao targets enabling L7 dispatch in hardware. L7 dispatch is a representative type of L7 processing, as all software proxies—regardless of how they are configured or what types of L7 processing they implement—must choose where to forward the messages.

In this chapter, first, we show how costly the typical L7 processing in software like L7 dispatch is and analyze the sources of the overhead. Then, we discuss the challenges of offloading such L7 processing to the high-speed packet-processing pipeline like RMT [73]. Last, we detail the design, implementation, and evaluation of QingNiao, to see how it works to address the challenges and how it can be generalized to other L7 processing beyond L7 dispatch.

4.1 The case for L7 dispatch in hardware

As depicted in Figure 4.1(a), L7 dispatch requires the processing elements (*e.g.,* software proxies) to look at the application-level (L7) data in each message, and forward them to the downstream elements (*e.g.,* typically processes) based on the configured L7 rules (*e.g.,* matching against the configured URL patterns [34]). L7 dispatch impacts the throughput and latency almost for every message: it is reported that Istio [24]—the most commonly used service mesh—imposes

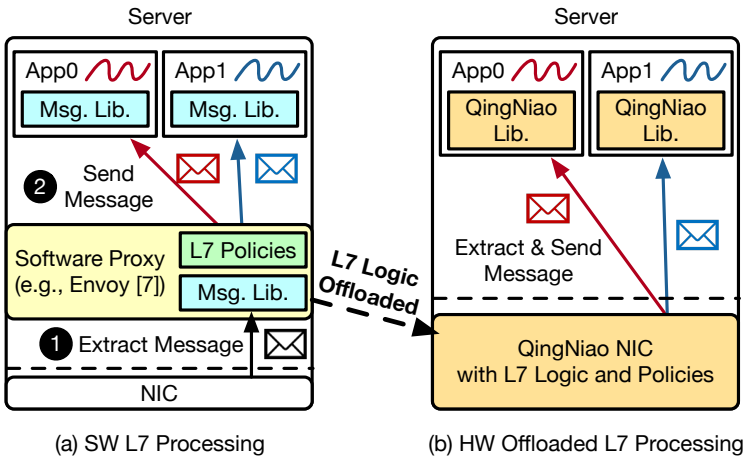


Figure 4.1: Architectural comparison of L7 processing implemented in software and hardware (*e.g.,* our solution QingNiao).

significant overhead for this operation: it increases latency by 269% [143] and CPU usage by 163%. Consequently, a lot of work [14, 24, 23, 26, 34, 125, 51, 31, 86, 6, 32] has aimed to improve dispatch performance.

However, none of these efforts can bypass the communication pattern in the current architecture (Figure 4.1a). This is because of two steps that are required for software dispatch: (1) extracting application messages from network packets and applying policies (*e.g.,* authentication) on the extracted data; (2) using inter-process communication (IPC) to send the application message to the thread that processes it. Both steps add overheads.

To quantify it, we start by measuring the overheads on response latency for performing L7 dispatch in software. To do so we compare response latency for FastHTTP [16] when running standalone to a deployment where Envoy [14],[1] a widely adopted L7 proxy, is used to implement L7 dispatch. We use the same FastHTTP configuration in both cases. For the Envoy-based deployment, we use one Envoy instance and two FastHTTP instances, and pin Envoy and FastHTTP to different cores. In both cases, we ensure there is no request queuing and report average latency across 100K iterations. The experimental setup is detailed in and consistent with §4.6.

[1] We used version v1.50.0 and v1.21.0 for FastHTTP and Envoy, respectively.

Component	Time
HTTP Processing	27 µs
Interprocess Communications	15.23 µs
Total	42.23 µs

Table 4.1: Dissection of additional latency brought by Envoy.

In our setting, FastHTTP when run without Envoy has a response latency of 45 µs. As shown in Table 4.1, running FastHTTP with Envoy increases this latency to 87.23 µs, an increase of 91.9%. We used BCC's funclatency [15] to understand the source of these overheads. We found that Envoy's HTTP parsing contributed approximately 27 µs, while IPC between Envoy and FastHTTP added approximately 15.23 µs. To put this breakdown in context, HTTP parsing within Envoy takes 60% of the time FastHTTP takes to process a request, while IPC takes 33.8% of the time FastHTTP takes for processing requests. Thus both contribute to the additional response latency.

Not only does Envoy increase response latency, it also decreases throughput. We measured this using wrk [60], and found that adding Envoy reduces throughput by over 9×: FastHTTP without Envoy can process 123.65 Krps, while with Envoy it can only process 13.13 Krps.

High-performance software dispatcher. Software solutions, including Shenango [116], Shinjuku [102], Caladan [85] and Junction [84] aim to reduce the overheads of software dispatch, however they only allow L3/4 information to be used for dispatch. Our evaluation (§4.6.1) shows, extending them to consider L7 information adds significant overhead, and negates much of their performance advantage.

Hardware dispatchers. Recent work has also developed hardware-accelerated dispatches. However, many of these, including RackSched [141] and RingLeader [111], only allow for L3/4 dispatch rules, and thus do not apply to our setting. Other work, in particular, Cerebros [122] and Nebula [132], move a single application's RPC logic into the NIC (note, both use the term 'dispatch' to refer to code dispatch, *i.e.,* calling the appropriate function for an RPC message, a sense that is

different than our use of the term). However, these approaches cannot implement many common proxy policies, *e.g.,* they cannot distribute RPCs across shards of the same service.

4.2 Challenges to offloading L7 dispatch

To mitigate the overheads of L7 dispatch and improve performance, we are seeking to delegate L7 dispatch to the NIC hardware which is located on the application's communication path, as shown in Figure 4.1(b). Potentially, the L7 dispatch logic is directly implemented on the NIC, thus avoiding expensive L7 processing alongside IPC to be performed on the CPU host.

However, it is challenging to directly extend the existing packet-processing pipeline like RMT [73] due to the following unique characteristics of L7 processing: The first, was a mismatch between the size of application messages and network packets: an application message can be several megabytes in size, exceeding the MTU of most networks. Consequently, a single application message is often *segmented*, which is split across multiple packets. Software-based proxies rely on the OS network stack to reassemble packet payloads into an application message before processing. However, implementing segmentation reassembly—*e.g.,* handling recovery of packet loss, buffer management, etc.—on hardware requires complicated logic and large amounts of memory [119].

The second, was the presence of variable-length fields (*e.g.,* strings or vectors) within the application messages. Existing match-action pipelines are designed to process fixed-length fields (*e.g.,* IP addresses), and this affects both how match rules are specified (using exact values or bit masks) and processed.

These two unique features of L7 processing make offloading to existing packet-processing pipelines like RMT [73] impossible as discussed in Chapter 2. This is due to the fact that RMT-like pipelines are designed to extract, store, and modify the packet header fields with limited on-chip memory, which makes it essentially hard to support the variable-length L7 fields. To this, we design

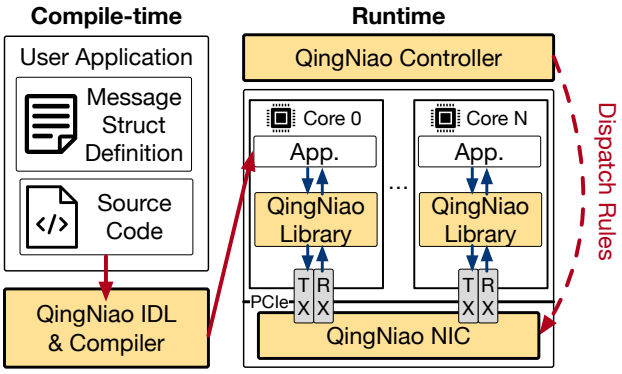


Figure 4.2: QingNiao's design consists of the colored components.

new hardware primitives (§4.4.2) in QingNiao.

4.3 QingNiao Design

QingNiao's goal is to offload L7 dispatch to a NIC, and thus eliminate the overheads (§4.1) of software dispatchers. We had two primary goals when designing QingNiao: (1) *generality*, allowing QingNiao to be used by different L7 message definitions and with different L7 dispatch rules; (2) *hardware resource efficiency*, allowing QingNiao to be implemented on a wide variety of NICs and combined with other types of offloads.

At a high level, to address the challenges (§4.2), we design a new encoding scheme (§4.4.1) in QingNiao to avoid requiring reassembly on the NIC: each Menshen packet contains a *message ID*, and the first packet of a message contains all the fields that are used to dispatch the message. This design allows the NIC to process individual packets without reassembly: the NIC determines and caches a dispatch decision when it receives the message's first packet, and all subsequent message packets are dispatched using the cache. Additionally, we adopt a skip-and-match based rule specification (§4.4.2) to specify matching rules over variable length fields, and design a hardware match engine that can efficiently match packet contents using these rules without storing the interested L7 fields.

QingNiao is designed to operate in internal services, where applications running atop can be easily linked to QingNiao library. External messages can be translated via gateways [31]. It consists of the three components (Figure 4.2): (1) **QingNiao software (§4.4.1)**, which provides interfaces for the applications: a QingNiao Interface Definition Language (IDL) and its compiler to customize L7 message structs for achieving *generality*, a runtime API library to interact with the underlying QingNiao NIC; (2) **QingNiao hardware (§4.4.2)**, which achieves *resource efficiency* by buffering no packets, thus avoiding excessive memory footprint and complex on-NIC buffer management for L7 message reassembly; (3) **QingNiao controller**, which manages on-NIC QingNiao L7 dispatch rules at runtime.

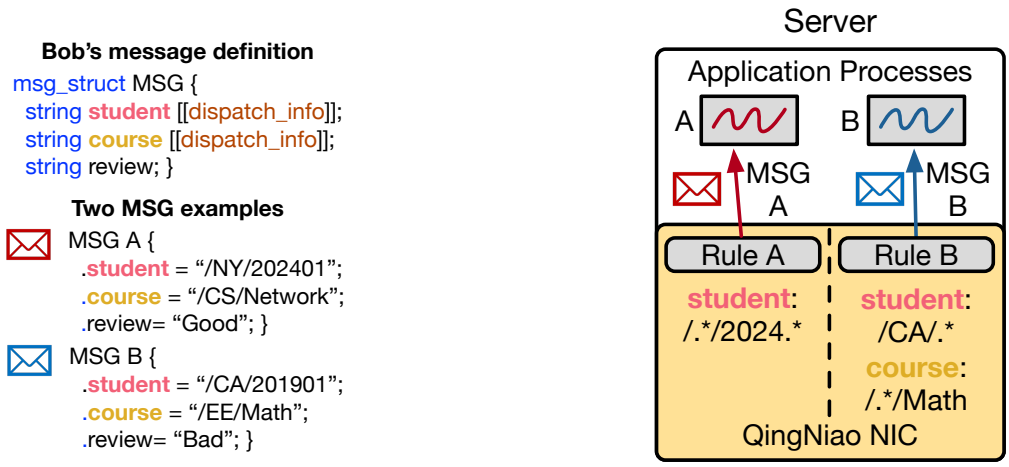
Data layout is the core technique that enables QingNiao. When segmenting an application message that spans multiple packets, QingNiao lays out data so that: (1) portions relevant to L7 dispatch appear in the message's first packet, (2) and each packet carries this message's unique ID. As a result, QingNiao allows to dispatch every packet: a dispatch decision can be made on the message's first packet, which can be looked up for subsequent packets using the same message ID.

To make prototyping feasible, our current design does not use TCP as a transport protocol, and instead uses a simpler QingNiao protocol (QNP) described in §4.5.2. As we discuss later in §4.8, our design choices can be used with TCP or other transport protocols.

4.4 An Example of QingNiao

Before diving into the details of QingNiao, we first present an example of how to build and deploy an application atop QingNiao. We use this example throughout the rest of this section to illustrate QingNiao's design.

Suppose Bob, a developer at a university, wants to build and deploy a course review application. The application needs to be able to track reviews for courses offered by all of the university's departments. Bob starts by defining his application's interface. The application processes messages



(a) Example of messages.

(b) Example of dispatch rules.

Figure 4.3: Example of the message struct definition and dispatch rules.

(Figure 4.3(a)) that consist of three variable-length string fields: (1) *"student"* with the format "/<campus>/<id>"; (2) *"course"* with the format "/<department>/<course>"; and (3) *"review"*.

Bob wants to ensure that the application can scale to all users at the university, and span multiple cores. His design for scaling requires that he launches multiple processes that execute the application, and shard reviews (*e.g.,* by student ID or department) across these processes. Because data is sharded, he must ensure that messages are forwarded to the correct shard, and he must use an L7 dispatcher.

To use QingNiao as an L7 dispatcher, Bob first defines the application messages (Figure 4.3(a)) using the QingNiao IDL (§4.4.1). When defining application messages, Bob uses the annotation `dispatch_info` to indicate the set of fields that can be used for L7 dispatch. Then, Bob writes the course review application using the QingNiao library (§4.4.1). Next, Bob must decide what L7 dispatch rules he should use. We show two example dispatch rules in Figure 4.3(b): Rule A directs any messages where the *"student"* field matches the pattern `/.*/2024` to process A, while Rule B directs messages where the *"student"* matches `/CA/.*` and *"course"* field matches `/.*/Math` to process B.

After creating the application and determining dispatch rules, Bob deploys all of the application

processes on a server equipped with a QingNiao NIC. Then Bob uses QingNiao's controller (Figure 4.2), to configure dispatch rules. Once this is done, QingNiao forwards messages to the configured processes, thus performing L7 dispatch.

4.4.1 QingNiao Software Stack

We discuss the design of QingNiao software and defer the detailed implementation to §4.5.2.

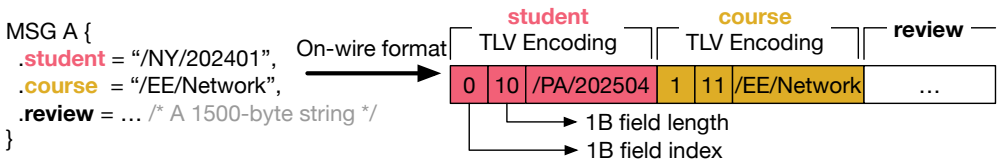
QingNiao IDL and compiler. Similar to protobuf [43], QingNiao provides an IDL and compiler for customizing message structs. Additionally, it provides a *dispatch_info* attribute specifier to mark fields used for L7 dispatch as in Figure 4.3(a).

Application Programming Interfaces (APIs). QingNiao provides a minimalistic programming model to applications, namely *send_msg()* and *recv_msg()* APIs in QingNiao library.

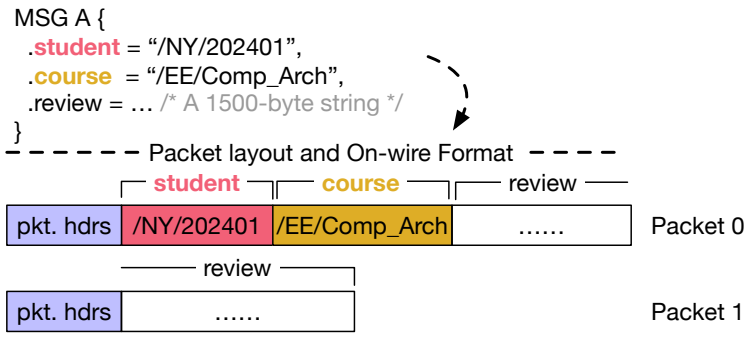
To send messages to the underlying network, applications use the *send_msg()* API, which lays out and serializes message fields. Specifically, for the layout, QingNiao library arranges the message fields such that the fields marked with "*dispatch_info*" appear before the other fields. Then the message is segmented into multiple packets if necessary, and the "*dispatch_info*" fields are guaranteed to appear complete in the first packet. In the packet header, each packet will carry the message's unique ID and its in-message packet sequence number. As we will show in §4.4.2, QingNiao hardware offloads L7 dispatch by leveraging this packet layout to avoid on-NIC packet buffering and message reassembly.

For the serialization, QingNiao library uses the serialization functions—generated by IDL compiler—to translate message fields into a Type-Length-Value (TLV [56]) format as in Figure 4.4(a), which is effective in expressing variable-length L7 values and widely used [43]. We elaborate the format and encoding in §4.5.2.

On a server equipped with QingNiao NIC, each application process is allocated with a set of TX/RX queues (Figure 4.5, §4.4.2). To receive messages from the underlying network, applications



(a) TLV format.



(b) Packet layout.

Figure 4.4: Example of QingNiao on-wire format for a message struct with two fields, where fields ‘student’ and ‘course’ are used for dispatch.

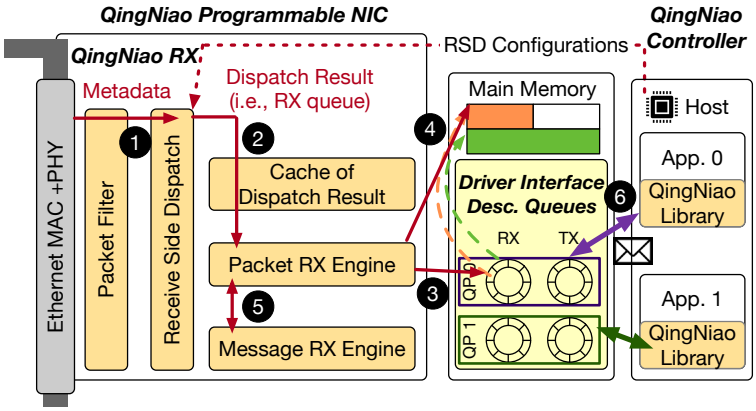


Figure 4.5: QingNiao’s RX data path overview.

use the *recv_msg()* API. The QingNiao NIC and library will ensure the packets with the same message are all received in order and deserialized into the original message, which is delivered to the application.

4.4.2 QingNiao Hardware Architecture

The RX path is where all of QingNiao's logic resides. By contrast, the TX path works as any other NICs: it sends out packets provided by our library. Thus, we only describe the RX path in this section. We first detail the basic packet handling, and then describe the Receive Side Dispatch (RSD) module that implements the L7 dispatch logic.

As we explained above, when sending a multi-packet message, QingNiao library (1) places *dispatch_info* fields in the first packet; (2) adds a message ID to all packets. The QingNiao NIC uses this layout: when it receives the first packet for a message it uses the *dispatch_info* field to decide where the packet should be dispatched. It then caches this decision, and uses the message ID field to find the cached result and dispatch subsequent packets.

This message layout allows us to avoid buffering packets and reassembling messages on NIC, reducing resource requirements [53, 13]. The only per-message state maintained by the QingNiao NIC are: descriptor for DMAing packet payloads, sequence numbers of received packets, timer for message expiration, and cached dispatch result.

QingNiao's RX path (Figure 4.5) performs following steps when it receives a packet:

- ❶ a packet filter parses out packet metadata (*i.e.,* message ID, length, and packet sequence number) and inputs it to RSD along with the packet to compute the ❷ dispatch result (*i.e.,* RX queue) for the message.
- The result is cached for this message ID and used by Packet Receive Engine to ❸ fetch a descriptor from the target RX queue, then ❹ packet content is DMAed to the host, by using the fetched descriptor and the packet's in-message sequence number to determine its offset in the descriptor-pointed memory region. QingNiao uses per-message descriptor and ensures the pointed memory region is large enough for the message itself.
- ❺ Message Receive Engine tracks the following states per message: the fetched descriptor,

an indicator of received and missing packets to show whether a message is fully received, and a timer when the last packet of the message was received. These states are essential for QingNiao's correctness and resource efficiency: the descriptor is kept for the subsequent packets within the same message, and the per-message timer is used to notify QingNiao to reclaim the corresponding on-NIC states and at-host memory once it expires.

- ❻ When all packets of the message are received, QingNiao NIC notifies the host of the successful message delivery. By calling QingNiao *recv_msg()* API, applications can directly receive the deserialized messages from the allocated RX queues.

We now discuss two core design decisions that we made: how we deliver messages to applications, and how we process L7 dispatch rules using our receive side dispatch module.

Separation of packet and message delivery. The NIC DMAs each received packet to host memory immediately. However, it delays notifying the host until all packets in a message are received, thus providing the host with an abstraction where it receives the whole message.

One concern with this approach is what to do if the second (or a later) packet for a message arrives before the first? Dispatch information is only carried in the first packet, and without this information we cannot decide where to DMA the packet. We currently handle such **out-of-order** packet arrivals by discarding any packets (for a message) that arrive before the first message packet. This simplification has been adopted in other NIC designs, including several RDMA NICs that use a go-back-N retransmission strategy [93] and Tesla's TTPoE, to limit resource requirements.

QingNiao Receive Side Dispatch. QingNiao Receive Side Dispatch (RSD) is the hardware module responsible for mapping messages to desired RX queues based on the specified L7 dispatch rules. We first describe the type of dispatch rules supported by QingNiao, and then elaborate how RSD is designed to match incoming messages against configured dispatch rules.

QingNiao dispatch rules based on skip-and-match. As discussed in §4.2, L7 dispatch usually requires matching message fields of byte strings against configured rules. To support this, one

straightforward way is to implement on-NIC regex as it is expressive for string matching [45].

Unfortunately, implementing reconfigurable regex matching based on Deterministic Finite Automatas (DFAs) using lookup tables [89] is complex and challenging [90] since the number of states of DFAs grows exponentially as the complexity of regex increases [70, 113].

To this end, QingNiao instead adopts a simpler string matcher based on a *skip-and-match* abstraction which is a relaxed form of regex. Following regex [45] convention, QingNiao's skip-and-match supports two primitive patterns: (1) *Skip*, *i.e.,* '`.Nstr`', where *N* bytes are skipped and a byte string `str` is matched; (2) *SkipUtil*, *i.e.,* '`[^/]*/`', which bypasses any bytes until a configured '`/`' is matched. This allows QingNiao to match against *fixed* or *variable* number of bytes of the input. The rationale behind this is that dispatch rules often match on only portions of a string, *e.g.,* the first few bytes of an argument or a URL pattern. Specifically, one skip-and-match consists of an alternating sequence of bytes that should be skipped and strings that should be matched.

Recall Bob's dispatch Rule A (*i.e.,* `/.*/2024`) in Figure 4.3(b) as an example, it can be achieved by two skip-and-matches: (1) *Skip* 0 bytes and Match `/`; (2) *SkipUntil* a specified character (*e.g.,* '`/`') and Match `2024`. This rule matches any string that has the prefix starting with `/`, and ending with `/2024`. Concretely, it will match the string `/CA/20240919`, but not the string `/PA/202345`.

In practice, skip-and-match is expressive enough to encode L7 dispatch rules given a known string format. For example, in an HTTP URL [57] string, a path component typically consists of a sequence of path segments separated by a slash (*e.g.,* `https://harrypotter.fandom.com/wiki/Hedwig`). Using skip-and-match's *Skip* and *SkipUntil* primitives—two SkipUtils bypass the domain ('harrypotter.fandom.com/wiki') and one Skip matches 'Hedwig'—can easily match desired portions of such strings as elaborated before.

How RSD works. QingNiao RSD extracts L7 message struct fields from packet contents and matches them against the configured dispatch rules to emit the dispatch result (*i.e.,* RX queue). To fulfill this, RSD should be able to (1) decode TLV-encoded message fields, which requires RSD to parse field type, length, and value in order; (2) move along the field's byte sequence and match it against

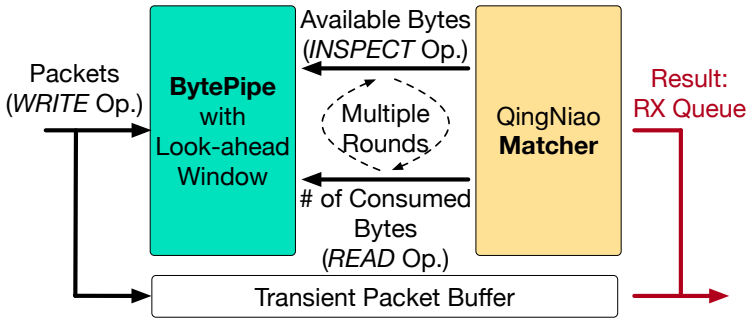


Figure 4.6: Receive Side Dispatch's components: (1) BytePipe with look-ahead window; (2) programmable Matcher which implements skip-and-match.

the dispatch rules, consisting of skip-and-matches. They both require processing bytes serially. However, memory elements typically do not natively support serial access at byte granularity [65].

Therefore, the RSD (Figure 4.6) is designed with 2 parts: (1) a BytePipe, and (2) a programmable Matcher. BytePipe takes in the packets and serves as a serial transient byte stream for the Matcher as it presents the Matcher with the available bytes. Matcher will consume bytes from BytePipe to parse L7 message fields in the TLV format, and match them against configured dispatch rules. The original packet contents are also stored in the transient packet buffer, which will emit the packets along with the dispatch result to the downstream module.

Specifically, as shown in Figure 4.6, to support serial parsing for the Matcher, BytePipe provides 3 operations: *read* and *write*—with a number of bytes as an argument—specify how many bytes are read out and written in, respectively; *inspect* checks bytes of a look-ahead window. By interacting with BytePipe, Matcher is designed to carry out skip-and-match: it uses *read* operation to *skip* bytes and then instructs *inspect* operation to get a window of bytes, which is used to *match* against the bytes configured in dispatch rules.

Additionally, when QingNiao control plane starts to reconfigure dispatch rules for an application, it will also notify the packet filter to discard the traffic from the applications under reconfiguration. This allows Matcher to support disruption-free reconfiguration without impacting existing traffic from other running applications, as shown in §4.6.2. In §4.5.1, we describe an FPGA

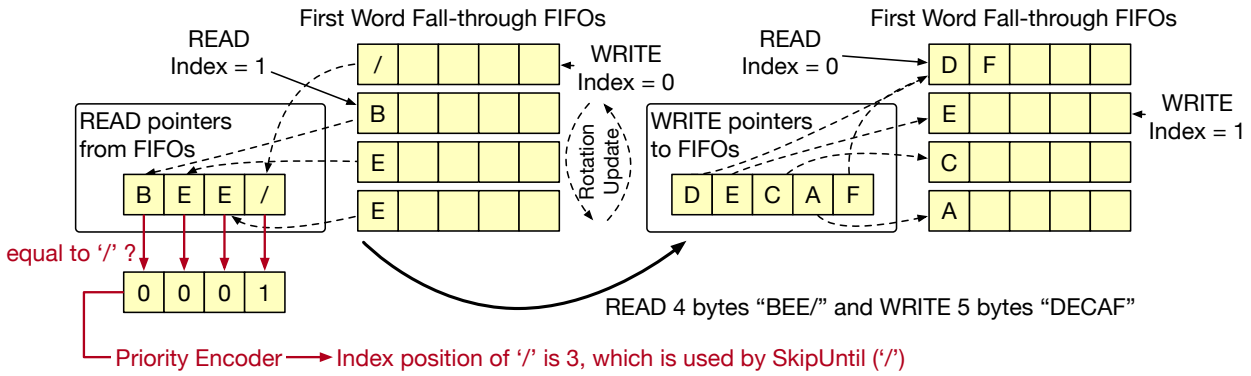


Figure 4.7: A BytePipe example with read and write index being 1 and 0 at the beginning, respectively. A *read* operation reads out 4 bytes (*i.e.,* "BEE/") and a following *write* operation writes in 5 bytes (*i.e.,* "DECAF"). BytePipe leverages a priority encoder to indicate the position of the specified character used by SkipUntil.

implementation of RSD.

4.5 QingNiao Implementation

In this section, we detail the prototype of QingNiao hardware and software stack. We also describe two optimizations that our implementation uses to improve the NIC's throughput.

4.5.1 QingNiao Hardware Prototype

We prototype QingNiao hardware using FPGA and integrate it into Corundum [83]. Our NIC implementation consists of 6004 lines of Verilog, and runs at 250 MHz (which allows us to hit line-rate on our NIC). We discuss the three main modules of our implementation below:

Receive Side Dispatch. As discussed in §4.4.2, RSD mainly consists of *BytePipe* and *Matcher* modules, of which we show the implementations in the following.

BytePipe. As in Figure 4.7, inspired by protoacc [104], BytePipe is implemented as a set of *parallel* First-Word Fall-Through FIFO queues (FWFT FIFOs), where the word size is 1 byte and the first byte at the head is immediately presented if it is not empty. This allows BytePipe to accept a simultaneous read/write of a sequence of bytes, which is capped by the number of parallel FWFT

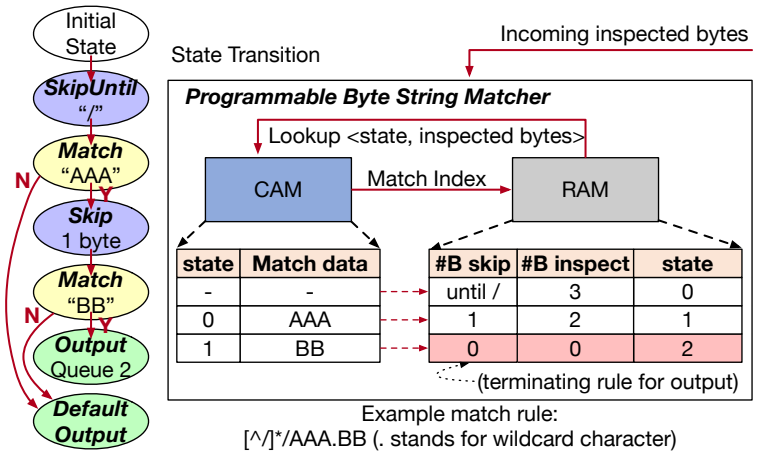


Figure 4.8: Matcher implements the skip-and-match. Skip and Match are encoded in RAM and CAM entries respectively. RAM entries with skipping and matching 0 bytes denote exit states: it shows the output queue index in the last column.

FIFOs. To support *read*, *write* and *inspect*, BytePipe keeps track of *read* and *write* index when executing *read* and *write* operations, respectively.

Take Figure 4.7 as an example. For **read**, BytePipe starts with the *read* indexed FIFO and iterates on parallel FIFOs to extract byte till the desired length. A 4-byte read operation starts from 1-indexed FIFO and BytePipe outputs `BEE/`. Note that reading from parallel FIFOs can be done simultaneously. **Inspect** works in the same way as *read*, except that it only exposes the bytes but does not consume them. At the same time, to support *SkipUntil*, a priority encoder is used to indicate the index of the specified character (*e.g.,* '/' in the example), which can be used as the number of bytes to be skipped. For **write**, BytePipe arranges the insertion of the byte to each FIFO starting from the *write* indexed FIFO. As shown in Figure 4.7, `DECAF` of 5 bytes will be spread out over all 4 FIFOs and two rounds of inserting bytes to FIFOs, where the 0-indexed FIFO will be written twice.

In our current prototype, the number of the parallel FWFT-FIFOs is set to 64, allowing it to support at most 64 bytes for each round, where each operation may take multiple rounds. BytePipe needs 3 cycles for the *read* and *write* round, while *inspect* operates immediately. The depth of each FIFO is set to 128, allowing it to store 8192 bytes at most transiently.

Matcher. We take inspiration from implementing hardware packet parsers [89] to implement *skip-and-match* in Matcher by mapping *skip*s and *match*es to Random Access Memory (RAM) and Content Addressable Memory (CAM) respectively. Specifically, as in Figure 4.8, a dispatch rule consisting of skip-and-matches can be translated into a state machine, where states alternate between skipping bytes and matching strings until a final output is found. We map states that skip bytes into entries in RAM, and states that match string into entries in a CAM so that the string matching can be implemented as a content-addressable operation. Rule matching then requires going back and forth between these two units, and Matcher outputs an RX queue identifier when a match/mismatch has been found.

To meet timing at 250MHz (*i.e.,* 4ns), the number of entries for configuring dispatch rules is set to 512 for both RAM and CAM in our current prototype. The RAM entry is designed to be 32-bit width: an 8-bit field index to be matched against, two 8-bit numbers of how many bytes to skip and match for one skip-and-match, and an 8-bit state number to represent the translated state machine of corresponding dispatch rule in the Matcher. In our implementation, we use the number of skipped bytes set to 0xff to indicate the *SkipUntil* primitive, which means to skip to the character '/'. Otherwise, it is configured as the *Skip* primitive. We ensure the number of skipped bytes in each skip is smaller than 64, as BytePipe can support reading 64B at most simultaneously.

Each CAM entry is 96-bit wide: a 24-bit number to be matched against an 8-bit application ID, 8-bit message type and 8-bit field index, an 8-bit numbered state and an 8-byte (*i.e.,* 64-bit) byte string to be matched against. The terminating rule is encoded into RAM entry of skip-and-match with 0 bytes to skip and match.

Cache of dispatch result. We use a small RAM as a stash to cache the dispatching result for each message. We simply assign the available stash entry to the incoming message, otherwise, entires are evicted in an LRU manner. Specifically, each stash entry is 41-bit wide: 8-bit cached dispatching result, 32-bit corresponding message ID, and 1-bit indicator of validity. The depth of the stash is set to 128.

QingNiao hardware states. As described in §4.4.2, instead of buffering packets for L7 message reassembly on NIC, QingNiao only tracks neccesary states for in-flight messages. In our current prototype, the number of tracked message entries is set to 512. The states of each entry consist of a 16-byte descriptor for DMAing packets from NIC to host, a 24-bit timer, and an 8-bit tracker of received individual packets within the message. The timeout for invalidating existing message entries is fixed to 1s.

4.5.2 QingNiao Software Prototype

```
1 struct qnp_pkt_hdr {
2   u8  app_id; // application ID
3   u8  msg_type; // indicator of message type
4   u32 msg_id; // message ID
5   u32 msg_acked_id; // acked message ID
6   u8  msg_len_in_pkts; // # of pkt in msg
7   u8  pkt_seq_num_in_msg; // pkt's seqnum in msg
8   u8  pkt_flag; // indicator of DATA or ACK
9   // for optimization
10  u8  seg_cnt; // #segments that count for RSD
11  u8  paddings[PAD_SIZE]; // for alignment
12 };
```

Figure 4.9: QNP packet header definition.

QingNiao IDL compiler. QingNiao IDL currently offers *string* as the only type for the message struct fields—which is enough to represent variable-length byte string—and it assumes no nested message struct definition. We based our QingNiao IDL compiler on a protobuf [43] frontend parser. The QingNiao IDL compiler is written in 180 lines of Golang. It takes the message format as input and emits the corresponding data structure and (de)serialization functions in C++.

QingNiao protocol. For simplicity, instead of directly modifying the complex protocols (*e.g.,* TCP [47], QUIC [48]), we propose a QingNiao protocol (QNP) for QingNiao library to easily lay out the message packets as desired. QNP is designed to provide minimum support of reliable delivery by ACKing every message. Primarily, the QNP header consists of three main fields— a unique message ID, an in-message packet sequence number, and a message length in the number

of packets—which allows both sender and receiver to keep track of individual packets of each message. Figure 4.9 details the definition of QNP header.

We implement QNP on top of UDP packets. To simplify hardware implementation, we make sure the packet header (*i.e.,* Ethernet, IP, UDP, and QNP) is 512-bit aligned, which is the data width in QingNiao hardware. Thus we set *PAD_SIZE* to 8 in Figure 4.9. QNP's control loop works at the granularity of message, meaning that once an application message is received, QingNiao library will piggyback the ACK to the response. Compared with ACKing every individual packet at NIC, it avoids packet amplification that may lead to excessive traffic load on NIC's TX path.

QingNiao on-wire format. QingNiao uses a TLV format—which can effectively express variable-length strings (*e.g.,* protobuf [43], HTTP/2 [46])—to represent on-wire message struct fields in the QNP payload. Recall the example in Figure 4.4(a), a student field "/PA/202504" is encoded as 12 bytes: 1-byte field index, 1-byte field length, and 8-byte original string. Recall that we also encode a 1-byte field index in the RSD's Matcher configuration, the TLV's field index and length are used by QingNiao RSD to ensure dispatch rules are correctly enforced on the target struct fields by matching the field index and check the already parsed field byte length.

QingNiao library and driver. QingNiao library implements QingNiao APIs (*i.e., send_msg()* and *recv_msg()*) with QNP. Specifically, *send_msg()* API serializes the message into on-wire bytes using the serialization function generated by QingNiao IDL, segments the message into 1500-byte packets, and attaches them with a unique message ID per sender and a corresponding packet sequence number within the message (Figure 4.4(b)). Also, fields marked with *dispatch_info* are guaranteed to be laid out only in the first packets as described in §4.4.1. These packets of each message are then pushed to the QNP layer, which is responsible for reliable delivery. Once a message is ACKed, the corresponding memories used by its packets are reclaimed at the sender's QNP layer. Currently, QingNiao assumes each message has at most 4 packets. *recv_msg()* API grabs the received messages from the QNP layer.

We implemented the QingNiao library on top of DPDK 21.11 [12], and wrote a poll-mode driver

to allow applications to interface with the QingNiao NIC. The QingNiao software components are written in C++ and consist of 2757 LOC.

4.5.3 Optimizations of QingNiao

Parallel RSDs. Since Matcher needs to instruct BytePipe based on the configured dispatch rules, it requires more cycles to dispatch a packet as the complexity (*i.e.,* number of skip-and-match) increases. This will bottleneck the NIC processing. To alleviate the potential performance bottleneck due to RSD's serial processing, we apply parallel RSDs to process the incoming packets. Messages are sharded across parallel RSDs by their message IDs. A larger number of parallel RSDs improves performance but results in larger hardware resource consumption. Our current implementation uses four parallel RSDs, allowing us to meet our 250 MHz timing constraints.

Explicit indicator of dispatch info length. Additionally, when Matcher emits the dispatch result, it still needs to wait for BytePipe to finish flushing unused bytes of the same packet, to avoid polluting the processing of the next packet. However, flushing needs cycles. If it takes too long, the temporary packet buffer will be filled up, thus causing a bottleneck in the NIC pipeline. To this end, we introduce a field *seg_cnt* in QNP header (Figure 4.9) to indicate how many AXI-S segments, where the segment size is determined by the AXI-S data width (64B in our implementation), are used by Matcher when serializing dispatch_info fields (§4.5.2). Instead of inserting all the packet AXI-S segments, *seg_cnt* controls the number of AXI-S segments inserted to RSD's transient packet buffer, thus reducing the cycles for flushing unused bytes.

4.6 Evaluation

In this section, we evaluate QingNiao using integration with RocksDB [19] and conducting microbenchmarks. We compare QingNiao to both existing software and hardware solutions.

Experimental setup. As shown in Figure 4.10, We evaluate QingNiao on a server machine with

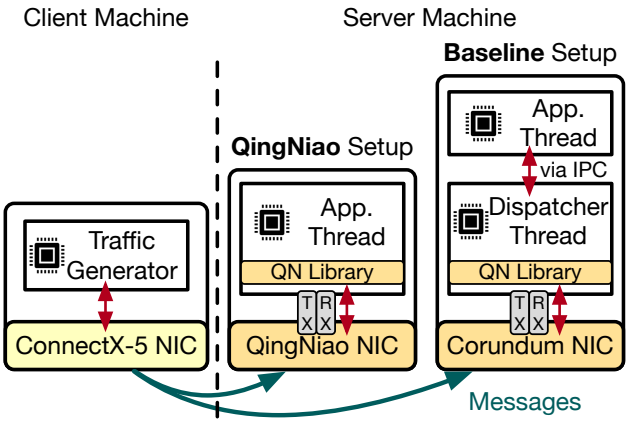


Figure 4.10: Experimental setup.

two Intel(R) Xeon(R) Gold 6132 14-core CPUs @ 2.6GHz, which runs Ubuntu LTS 20.04 with Linux kernel version 5.15.0. Additionally, this server is equipped with an AMD Xilinx Alveo U250 FPGA board [3] programmed as a NIC, which is plugged into a PCIe Gen3 x16 slot and is loaded with QingNiao hardware prototype and Corundum of the same base commit [10] for evaluating baselines. Our client machine is equipped with an Intel(R) Core(TM) i7-9700 8-core CPU @ 3.0GHz and a Nvidia Mellanox CX-5 100GbE NIC [36]. We disable hyper-threading on both machines and run processes within the same NUMA domain of the NIC to avoid expensive cross-NUMA communication. The NIC-to-NIC round-trip latency between two machines is ~800 ns in our testbed.

Baselines. We compare QingNiao to the following baselines, including both software and hardware solutions as described:

Software L7 dispatchers: For a direct and fair comparison,[2] we build software RSD dispatchers atop Corundum [83] using RSS [44] and IPC mechanisms to allow messages to be exchanged between the software dispatcher and application threads.[3] The IPC mechanisms between the dispatcher and application threads are (1) DPDK RTE Ring [12]; (2) an eBPF accelerated IPC mechanism similar

[2]The performance numbers of our baselines are much better than our tested numbers with Envoy (§4.2).

[3]The software dispatchers implement a straw-man match against the configured rules in a for loop in each skip-and-match.

to SPRIGHT [123].

We use a shared-nothing architecture: when one thread (*e.g.,* application, software dispatcher)—that interacts with the NIC—is launched, one free TX/RX queue pair is exclusively initialized and allocated to it. Throughout the evaluation, each thread is pinned on a unique core. The applications and software dispatches comprise 2396 lines of C++ code.

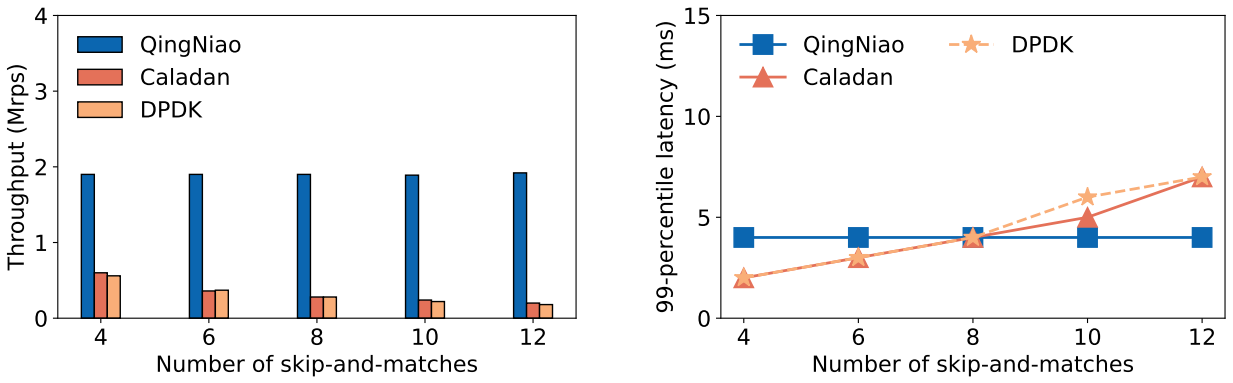
Caladan [85]: Besides, we also compare QingNiao to Caldan, a state-of-the-art system dispatching/scheduling application messages across processes. Since Caladan only supports L3/4 dispatch, to enable L7 dispatch in Caladan, we port software RSD implementation into Caladan IOKernel [85], which oversees every incoming message and distributes it to the applications running atop. To ensure a fair comparison, Caladan is always assigned one more dedicated core running its IOKernel. Caladan's performance numbers are reported using two CloudLab [9] d6515 nodes, where one server node and one client node are connected via a ToR switch. Each node is equipped with an AMD EPYC Rome 32-core CPU and a Nvidia Mellanox CX-5 100GbE NIC.

RingLeader [111] and RSS [44]: RingLeader offloads message scheduling to the NIC, but it does not support on-NIC L7 dispatch. Nevertheless, in §4.6.2 we compare QingNiao's performance (using L7 policies) to RingLeader[4] and RSS (both of which use L3/4 based policies).

Methodology. In §4.6.1, we integrate QingNiao with RocksDB [19], and compare it with software L7 dispatchers baselines. We demonstrate that QingNiao can (1) provide programmability to support dispatching rules; (2) benefit applications by improving throughput while decreasing latency as dispatching rules are more complex. Next, in §4.6.2, we use microbenchmarks to show that: (1) QingNiao achieves similar performance as L3/L4 hardware dispatchers; (2) QingNiao outperforms the L7 software dispatcher implementations. We also show QingNiao's hardware resource consumption.

Configurations of dispatch rules. As in §4.5.1, the total number of dispatch rules is constrained by the number of entries of RAM & CAM in RSD, which is 512 in our prototype. To show QingNiao's

[4]Our test uses the public RingLeader implementation [18] with default settings and VFIO [58] disabled.



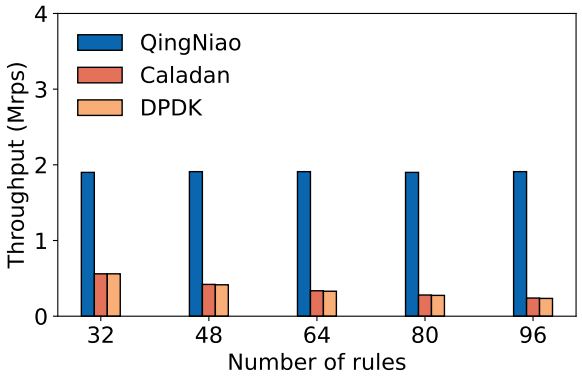
(a) Throughput comparison.

(b) Latency comparison.

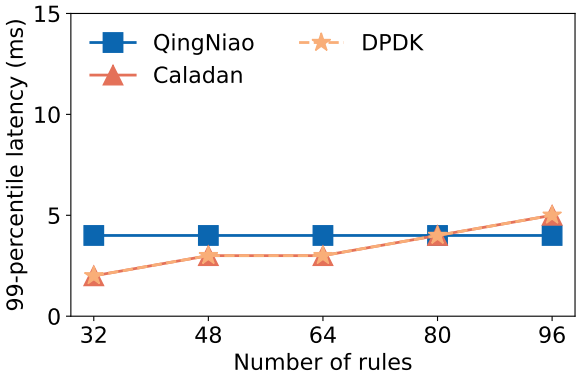
Figure 4.11: On throughput, QingNiao outperforms Caladan by 6.54× averagely with varying the number of skip-and-matches.

support of programmability, we explore 3 dimensions of QingNiao's skip-and-match (including both Skip and SkipUntil) abstraction: (1) the number of skip-and-matches; (2) the number of dispatch rules; (3) the number of matched bytes per skip-and-match,[5] where larger numbers mean that rules are more complex. When testing, we vary one dimension and keep the others fixed.[6] We use randomly generated skip-and-match rules, and use configuration that require fewer than 512 entries in the RSD RAM and CAM (to fit within available resources). Further, our workload ensures that each messages matches exactly one dispatch rule.

We use a closed-loop traffic generator written in DPDK 21.04 to generate our workload. The use of closed-loop generator allows us to measure the maximum throughput in the absence of drops. We configured the workload generator to ensure that it is not a bottleneck. When testing RSS, we use a workload that evenly distributes traffic across NIC queues.

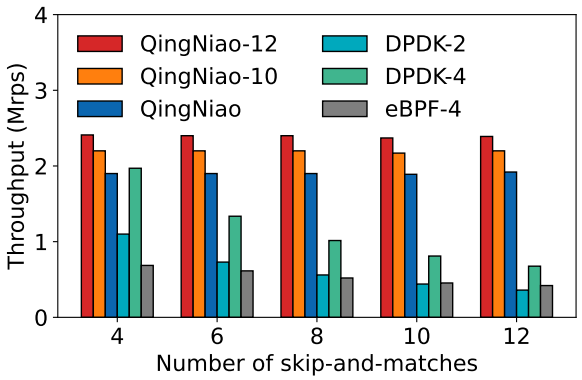


(a) Throughput comparison.

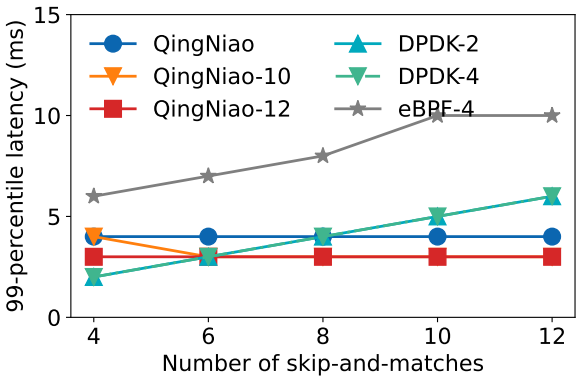


(b) Latency comparison.

Figure 4.12: On throughput, QingNiao outperforms Caladan by 5.67× averagely with varying the number of rules dimension.



(a) Throughput comparison.



(b) Latency comparison.

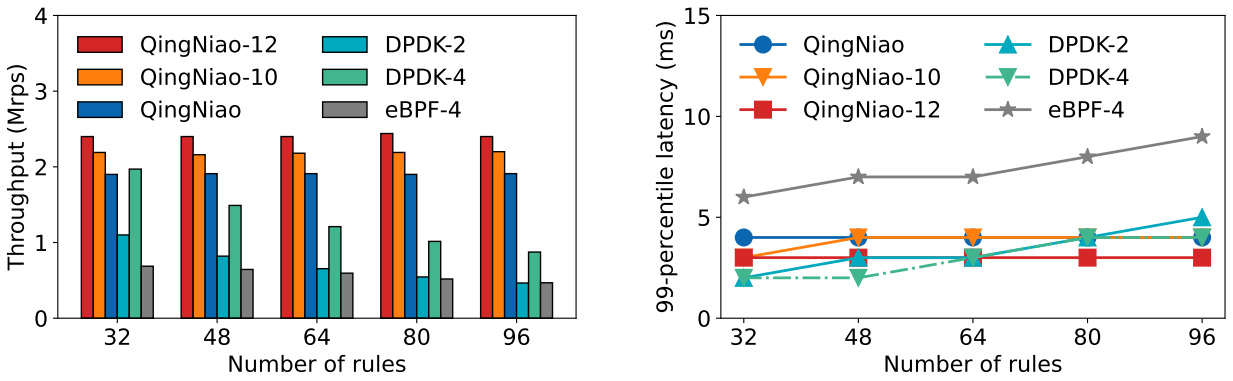
Figure 4.13: With the same number of cores, QingNiao outperforms software by 3.91× averagely with varying the number of skip-and-matches on throughput.

4.6.1 Integration with RocksDB

Workloads. We choose RocksDB [19] v7.9.2, a wide-deployed in-memory key-value store, as our application service to serve GET messages. We configure RocksDB to be backed by a 4G *tmpfs* folder. It pre-loads 3.14 million 64-byte keys with 64-byte values. Keys are evenly partitioned into 48 ranges, which are then equally assigned to each application thread. From the traffic generator,

[5]Results are presented in §4.6.2.

[6]The numbers of skip-and-matches, dispatch rules, and matched bytes per skip-and-match are fixed to 4, 32, and 5 respectively, when they are not the varying dimension.



(a) Throughput comparison.

(b) Latency comparison.

Figure 4.14: With the same number of cores, QingNiao outperforms software by 3.34× averagely with varying the number of rules dimension on throughput.

within each partition, message keys follow a Zipfian distribution with a Zipfian parameter equal to 0.9. The dispatch rules evenly map the keys of messages to the configured number of application threads. We test the maximal end-to-end throughput and report the 99-percentile latency, where numbers are gathered from the traffic generator and averaged for 5 runs.

Settings. In our baselines, we fix the number of application threads to 8. (1) First, to see how QingNiao's performance compares to state-of-the-art L7 dispatchers, we compare our performance to Caladan [85] and our DPDK L7 dispatcher. We allocate 1 core for the IOKernel [85] and DPDK dispatcher. (2) Secondly, we evaluate QingNiao's resource benefit by varying the number of cores used by the deployment: for software, the cores are allocated to both the dispatcher and application, while for QingNiao they are allocated to the application.

Figures 4.11 and 4.12 demonstrate that QingNiao is able to offload varied dispatch rules efficiently. In terms of achieved throughput, as we can observe from Figure 4.11(a) and Figure 4.12(a), QingNiao outperforms Caladan by 6.54× and 5.67×, respectively. The more complex dispatch rules are, the larger QingNiao's performance gain. This is because the CAM allows RSD to match multiple (*e.g.,* tens in our case) entries in a single cycle—unlike software dispatch. Both our DPDK baseline and Caladan have similar throughput: they both use DPDK to receive packets and shared

memory pipes for IPC [72]. However, the dispatcher logic is a throughput bottleneck in both cases.

Next, we increase the number of cores used in L7 dispatchers. We denote DPDK-*N* as the DPDK baseline using *N* cores for dispatchers, and denote QingNiao-*M* as the total cores used by QingNiao is *M*. When configured with same number of cores, QingNiao-12 outperforms DPDK-4 and eBPF-4 baselines by 2.44× and 4.88× (Figure 4.13(a) where we vary the complexity of rules), and 2.03× and 4.43× (Figure 4.14(a) where we vary number of rules).

In terms of end-to-end latency, as shown in Figure 4.13(b) and 4.14(b), QingNiao-12 reduces latency by 62% and 59%, respectively compared to eBPF-4. Further, our results show that software latency grows as rule complexity increases (Figure 4.11(b) and 4.13(b)), and also as number of rules increase (Figure 4.12(b) and 4.14(b)). By contrast, QingNiao's latency does not depend on rule-complexity or the number of rules.

Takeaway #1. Our RocksDB evaluation shows that for a real program, QingNiao can outperform state-of-the-art software dispatchers, achieving ~3× higher throughput, and ~ 60% lower 99-percentile latency.

4.6.2 Microbenchmarks

Throughout the microbenchmarks below, unless stated, the traffic generator generates 128-byte messages, including all the packet headers (§4.5.1), and waits for a 128-byte response from the server application thread. We term this application as PingPong, where each PingPong application has its unique message format and dispatch rules.

In the following, we have two scenarios: (1) *single* PingPong, where it compares QingNiao with L7-agnostic hardware dispatchers and QingNiao's software implementation; (2) *multiple* PingPong, where it demonstrates QingNiao's support of multiple concurrent applications. The results are collected at the traffic generator and averaged over 5 runs.

We start by evaluating QingNiao using a *single* PingPong.

System	QingNiao							RSS	RingLeader
Number of S&M	1	2	4	8	16	32	48	-	-
Number of Cycles	15	21	33	57	105	201	297	3	25
Latency(ns)	448	469	517	613	805	1189	1573	-	-

Table 4.2: QingNiao's latency reported in simulation and testbed. QingNiao adds 6 cycles per skip-and-match (S&M).

Comparison against L3/4 hardware dispatchers. In terms of the *latency* of dispatching, L3/4 hardware dispatchers have a fixed number of cycles, since they are processing fixed-length fields (*e.g.,* IP addresses) at known offsets. In comparison, QingNiao's processing latency is dependent on the number of skip-and-matches. To show this, we report the number of cyles from a cycle-accurate simulator. Additionally, we timestamp the received packet at both the Ethernet MAC+PHY and the RSD modules (Figure 4.5) to measure the ingress latency—including all ingress processing (*e.g.,* packet filter, etc.)—on the testbed. We send 4K messages, with a gap of 50ms between messages to ensure there is no message queuing.

Compared with L3/4 dispatcher RSS and RingLeader [111] that take 3 and 25 cycles respectively to compute a dispatch decision, QingNiao incurs increasing latency as L7 matching becomes more complex as shown in Table 4.2. It reaches 297 cycles and leads to a total of 1573 ns ingress latency for 48 skip-and-matches.

We also evaluated the latency of an ASIC implementation by creating a prototype, with 4-parallel RSDs, using Synopsys DC [52] and FreePDK45nm [17]. The resulting ASIC runs at 1GHz and requires 0.766mm^2 chip area.

Next, we compare QingNiao's *througput* to RSS and RingLeader. For this evaluation we vary the number of cores (both number queues and application threads), and use simple skip-and-match rules that require 1 match. As shown in Figure 4.15,[7] despite the incurred additional

[7]RingLeader's performance numbers reported on our testbed are lower than the numbers reported in the original paper. This is because we use different hardware.

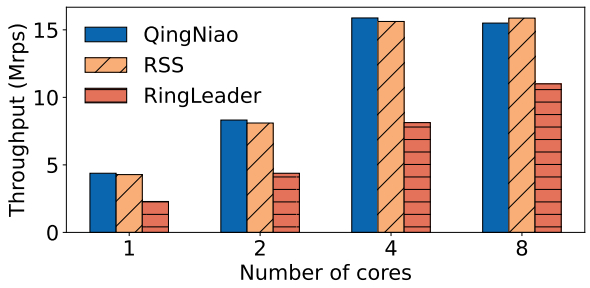


Figure 4.15: QingNiao achieves comparable performance as hardware L3/4 dispatcher.

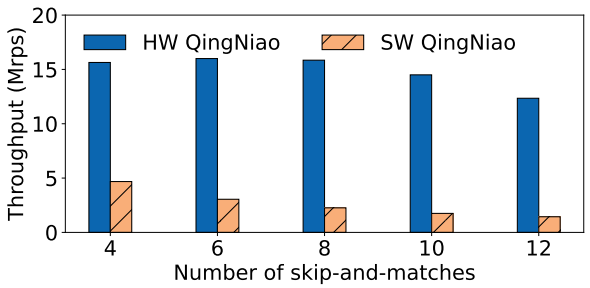


Figure 4.16: QingNiao outperforms software implementation by 6.49× averagely with varying number of skip-and-matchs.

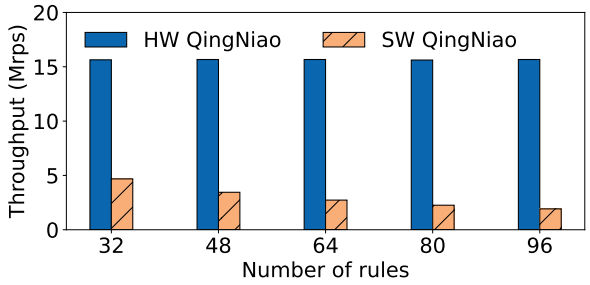


Figure 4.17: QingNiao outperforms software implementation by 5.74× averagely with varying number of rules.

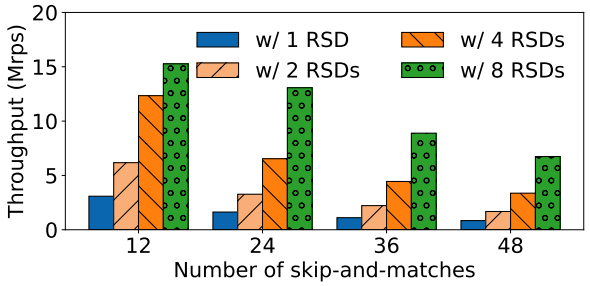


Figure 4.18: Having parallel RSDs improves throughput.

processing latency as we described, QingNiao achieves comparable throughput with RSS and can get 16.06Mrps for 4 cores. However, QingNiao's performance drops slightly to 15.34Mrps for 8 cores. This can be the result of more cache contention as the number of cores in the same NUMA domain increases.

Comparison against software dispatchers. We compared QingNiao to existing software dispatchers in §4.6.1. Here we compare against a software implementation of an RSD based approach. Our implementation runs the RSD logic in software, but rather than dispatching to another process it echoes it back to the sender (thus avoiding IPC overheads). We use 8 cores to run the software RSD logic. Our results, shown in Figure 4.16 and 4.17, QingNiao outperforms its software implementation by 6.49× and 5.74×, respectively. We do observe that latency for the hardware QingNiao implementation increases as the number of rules increase: when few rules are in use the NIC is

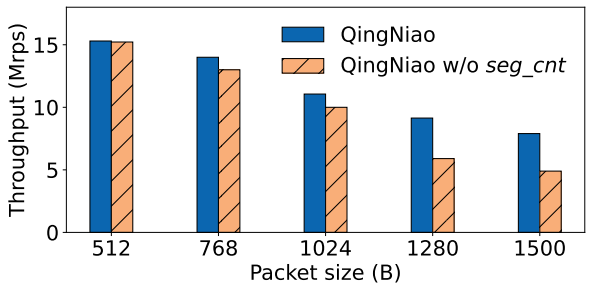


Figure 4.19: Introducing seg_cnt improves throughput.

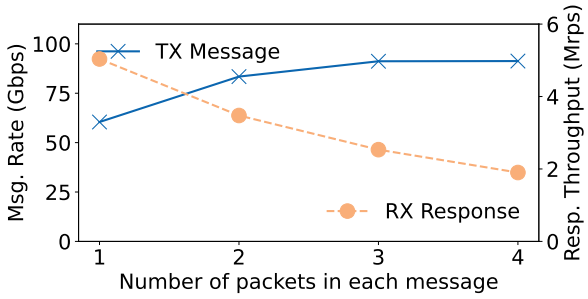


Figure 4.20: QingNiao supports multi-packet messages.

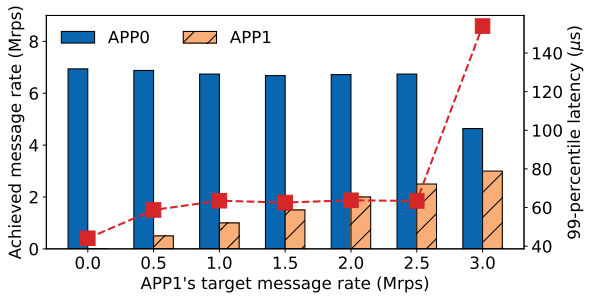


Figure 4.21: APP0's throughput decreases and latency increases as APP1's rate increases.

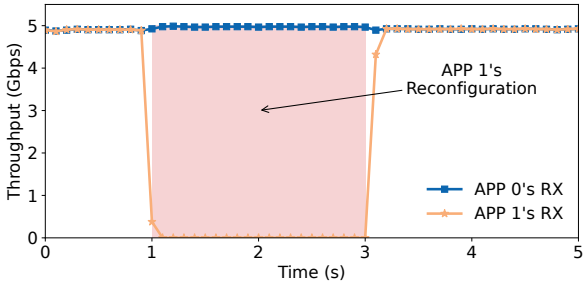


Figure 4.22: APP0's rate is not impacted when APP1 is reconfiguring in QingNiao at runtime.

bottlenecked on DMA, but as the number of rules increase the bottleneck moves to the RSD.

Number of parallel RSDs	LUTs as logic	LUTs as memory	BRAM
1	20033 (1.16%)	2644 (0.334%)	3 (0.112%)
2	40098 (2.32%)	5288 (0.668%)	6 (0.223%)
4	81589 (4.72%)	10576 (1.337%)	12 (0.446%)
8	163959 (9.45%)	21152 (2.674%)	24 (0.893%)

Table 4.3: FPGA resources usage for module of parallel RSDs.

Effectiveness of QingNiao's optimizations (§4.5.1) As discussed, when the number of skip-and-match increases, QingNiao's RSD needs more cycles to process one single packet, which stalls the entire NIC pipeline. Having multiple parallel RSDs alleviates this problem. As shown in Figure 4.18, the performance of QingNiao with 8 parallel RSDs doubles as compared to QingNiao configured with 4 parallel RSDs when the bottleneck shifts from DMA to RSD as the number of

skip-and-matches increases from 12 to 24. However, there is a tradeoff between performance and resource consumption. As shown in Table 4.3, the FPGA memory usage doubles as we double the number of RSDs.

The *seg_cnt* field in the HP header also helps improve QingNiao performance as shown in Figure 4.19. The intuition behind this is that the BytePipe can use this field to limit the number of bytes that it needs to buffer. This allows the RSD to examine fewer bytes and reach a dispatch decision sooner, leading to higher throughputs.

QingNiao supports multi-packet message. Finally, we evaluate QingNiao's performance when processing multi-packet messages. Figure 4.20 shows throughput when processing 4-packet messages (each packet is 1500 bytes) with 1 core: we observe a throughput of 91.28Gbps. This is the peak throughput we expected, and shows that accessing cached dispatch results does not add additional overhead.

Takeaway #2. QingNiao achieves similar performance compared to L3/4 hardware dispatchers (*i.e.,* RingLeader [111] and RSS) and outperforms its corresponding software implementation. Our optimizations are effective at improving QingNiao's performance, and QingNiao can support multi-packet messages without a loss of throughput.

Next, we evaluate QingNiao's performance when multiple applications are running on a single machine. Our evaluation runs two PingPong applications on the server.

Multiple applications with varied message formats. As we observed above, QingNiao's processing latency increases as the number of skip-and-matches increases. To show the impact of collocating messages with different processing latencies, we configure two different dispatch rules: we set the number of skip-and-matches to 2 (APP0) and 48 (APP1).

Running them separately gives us a throughput of 15.7Mrps and 3.37Mrps. APP0 is bottlenecked on DMA, while APP1 is bottlenecked on the RSD. To eliminate the DMA bottleneck, we fix APP0's target rate to 7Mrps and gradually increase APP1's target rate. As in Figure 4.21, when increasing APP1's rate from 0 to 2.5Mrps, APP0's achieved throughput decreases slightly

and latency increases. This is because QingNiao shards incoming messages into different RSDs by hashing message IDs, which delays APP0's processing. As APP1's rate reaches 3Mrps from 2.5Mrps, APP0's latency increases by 1.43×, since messages start to queue up at RSD processing. This is the result of QingNiao's design choice of being work-conserving rather than providing strong isolation between messages.

Support of disruption-free reconfiguration. We configure two applications to send messages at 5Gbps constantly. Individual response rates are measured at traffic generator with 0.1 s granularity. As Figure 4.22 shows, APP 1's rate drops to 0 when it starts to reconfigure at 1s, while APP 0's rate is still maintained at 5 Gbps. After this 1.5-s reconfiguration, APP 1's rate recovers to 5 Gbps, which demonstrates QingNiao's support of disruption-free reconfiguration at runtime.

Takeaway #3. The QingNiao implementation supports multiple applications: multiple applications can specify rules, and rule changes from one application do not disrupt another. However, QingNiao does not provide performance isolation between applications, and we discuss approaches for this next in §4.8.

Resource Usage of QingNiao. Finally, in Table 4.4 we report on the additional resources required by our implementation of the QingNiao NIC in contrast to Corundum: QingNiao uses about 4.92× more LUTRAM and 10.4% more BRAM than Corundum. Specifically, this additional resource usage mainly comes from implementing RSD, RAM, and CAM entries to store dispatch rules based on skip-and-match in QingNiao.

HW Resources	QingNiao	Corundum	Additional Usage
LUT	222622 (12.88%)	53723 (3.11%)	314.4%
LUTRAM	65038 (8.22%)	10978 (1.39%)	492.4%
BRAM	196.5 (7.31%)	178 (6.62%)	10.4%
URAM	10 (0.78%)	10 (0.78%)	0%

Table 4.4: FPGA resource usage of QingNiao vs Corundum.

Additionally, to show how the third dimension in §4.6—the number of matched bytes per skip-

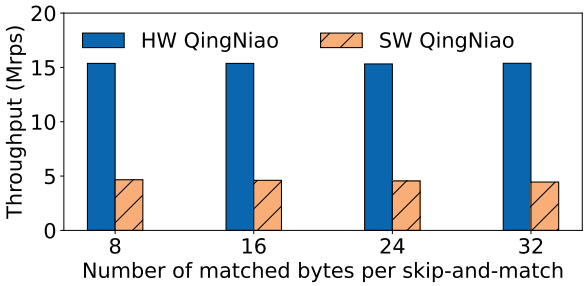


Figure 4.23: QingNiao outperforms its software implementation by 3.36× averagely with varying number of matched bytes.

and-match—impacts the performance, we slightly modify QingNiao implementation to support up to 32-byte matching per skip-and-match, by reducing the number of entries in RAM and CAM to run at 250MHz, which only impacts the configured number of dispatch rules but not impact QingNiao hardware performance. Here, we use 256-byte messages in the PingPong application (§4.6.2) for this test.

As shown in Figure 4.23, QingNiao's performance remains constant. This is because matching on CAM can support a longer sequence as long as it does not exceed CAM width. Also, the performance degradation of software implementation is negligible. The reason may be that matching on 32-byte sequences still does not pollute CPU cache to cause performance degradation.

4.7 Related Work

Message dispatching and scheduling. Recent works [85, 116, 77, 102] design software dispatching and scheduling mechanisms to better utilize CPUs. Dispatching is inherently orthogonal to scheduling: dispatching decides where the messages should go while scheduling determines in which order the messages should be served. MICA [109] dispatches messages using a client-assisted technique to reduce L7 dispatch overhead. However, it assumes single-packet messages and requires exposing underlying server configurations to the clients, which brings management concerns, *e.g.,* leaking server configurations.

QingNiao also offloads dispatch to hardware to save CPUs, similar to RingLeader [111] and RackSched [141], which only support L3/4 dispatch. Further, to handle multi-packet messages, both RackSched and QingNiao attach message ID to each packet, which alone is insufficient to support L7 dispatch based on message content. To this end, QingNiao additionally requires to explicit lay out dispatch information into the first packet of every message (§4.3).

Hardware acceleration for RPCs. Google's work [103] showed that a significant fraction of datacenter CPU cycles are spent processing RPC messages. Consequently, there have been several efforts to offload portions of this processing to specialized hardware, but focus on different problems compared to ours.

Protoacc [104], OptimusPrime [121], and Cereal [97] developed hardware offloads for serialization and deserialization, which is a common component of these overheads. These works are complementary to ours.

Another line of work (*e.g.,* Nebula [132], Cerebros [122]) that addresses the problem of offloading "dispatch". However, they address a *different* type of dispatch: **code dispatch**, where the hardware parses function ID from the packet header and calls a function to handle a received RPC message. By contrast, we address the **process dispatch** problem, i.e., we decide what process should receive a message. These problems are useful in different places: code dispatch is required when implementing RPC within a process, while process dispatch is used to shard messages across multiple instances of the same program. These problems also require different solutions: Cerebros (and software code dispatch approaches) use a lookup table that exact-matches a packet header field (*i.e.,* RPC type) to function, which can not be directly applied to matching variable-length message contents (*e.g.,* L7 fields) to decide where to forward the message as QingNiao (and process dispatch solutions) target.

4.8 Summary

In summary, this chapter presents QingNiao, a system combining both software toolchains and hardware primitives to enable offloading L7 dispatch to NICs. We have found that solely focusing on the NIC design can not achieve that because of current approaches for encoding multi-packet application messages, and limitations of how matching policies are specified and implemented in the hardware.

Consequently, we had to rethink our approach: QingNiao adopts a holistic design that uses a custom-made application message encoding that allows the hardware to process individual packets (rather than needing access to the whole message), an abstraction that is rich enough to specify policies over variable length fields efficiently implementable in hardware, and a software library that minimizes developer effort required to adopt QingNiao. We have prototyped QingNiao on an FPGA, demonstrating its viability.

QingNiao's lessons. Specifically, as discussed in §4.2, one inherent challenge of offloading L7 dispatch is that application messages (L7) and network packets (L4) do not strictly align with each other in today's protocols. As a result, the fields related to dispatch may start from arbitrary offsets of a message spanning multiple packets. This requires the NIC to buffer *all* received packets of the same message, scan for the location of dispatch-related fields, and can only release the message's corresponding allocated buffer until its related fields are processed by the L7 functions, which is complex and challenging for the hardware design (*e.g.,* buffer management).

Lesson: To enable L7 dispatch offloads without scanning and unbounded buffering in the NIC, fields related to dispatch must start at a known fixed offset of an early packet upon arrival at NIC. Consequently, two *necessary conditions* must be enforced by the L4 segmentation and L7 encoding: (a) at L4, application messages always start at the beginning of a packet but never the middle; (b) at L7, the fields relevant to dispatch are encoded and laid out at the beginning of the message. QingNiao demonstrates one feasible solution for offloading L7 dispatch.

QingNiao with existing transports. QingNiao changes how application messages are encoded and segmented as discussed. It is also feasible to use QingNiao with stream-oriented transports (*e.g.,* TCP [47], QUIC [48]), though it may need minor implementation changes on the sender side. Note, no protocol changes are required.

Specifically, the sender's protocol implementation would need to change in two ways (a) it must allow the application to packet align each application message, *i.e.,* ensure that each packet contains data from a single message; and (b) allow the application to determine how much (application) data is sent in each packet. Once this is done, implementing QingNiao on top of this modified transport implementation merely requires changing the application message format to follow QingNiao encoding, rather than the entire transport protocol.

On the receiver side, QingNiao can be integrated with transport offloads (*e.g.,* Beehive [110], Limago [124]) that leverage off-chip memory for buffering. Specifically, we can employ the technique like Indirect-TCP [69], which splits the NIC-to-Host communication into two: (a) one is between the on-NIC TCP module and QingNiao's RSD via network-on-chip substrates [110], which receives application messages; (b) the other is between RSD and host via DMA, which dispatches received messages to target processes. We leave such integration as future work. Similarly, QingNiao can also be easily applied to message-oriented protocols such as Homa [115] and MTP [130], where again only application messages need to accommodate QingNiao encoding.

End-to-end Encryption and QingNiao. End-to-end encryption (as provided by TLS [55]) can be a challenge for QingNiao, since it needs to operate on application data. We can adopt approaches like Google PSP [20] if encryption is desired. This approach would require administrators to use an out-of-band mechanism to configure (and update) cipher keys, but allow the NIC to decrypt packets before they are processed by QingNiao. Thus, it is feasible to deploy QingNiao-like approaches in environments where encryption is necessary.

5 | Conclusion

This dissertation presents two systems, Menshen and QingNiao, for making programmable networking devices (*i.e.,* RMT-based [73] devices) a wider adoption for application developers to fully reap the potential benefits they can provide.

At its core, Menshen designs isolation mechanisms—*i.e.,* hardware primitives in the form of small indirection tables—based on the techniques of space partitioning and overlays, thus allowing multiple application offloads to simultaneously run atop a single device without impacting each other. Additionally, Menshen also leverages the software techniques to enforce sanity checking to exclude potential malicious users.

QingNiao targets offloading L7 processing to the NIC and focus on one specific type of it, *i.e.,* L7 dispatch. To achieve that, QingNiao takes a holistic approach where it redesigns how the L7 messages are encoded, how the transport is implemented, and how the NIC hardware is architected. QingNiao demonstrates a feasible approach to offload L7 processing, which was thought too generic to be implemented on a packet-processing pipeline [76].

Both systems demonstrate that hardware primitives together with software toolchains are necessary to bring the programmable networking devices to the broader application developers. Also. to help further advance the research in this direction, we open source Menshen and QingNiao at [135] and [134], respectively.

5.1 Limitation and Future work

Here, we discuss other aspects which are worth further exploration.

Resource allocation. Currently, Menshen only supports statically partitioning hardware resources among different programs. However, the policy of resource allocation is important in a shared environment, since it can meet different properties (*e.g.,* fairness, etc.) by allocating the amounts of resources that can be accessed by a specific program. As we have shown in Menshen, a packet-processing pipeline is inherently a multi-resources environment, one may think of directly applying the Dominant-Resource Fair [88] policy. However, it may not work as different resources may correlate with each other. For example, one may consider TCAM and SRAM as two different resources, but in fact, one TCAM entry might correspond to different SRAM entries, which means that there is an implicit connection between these two types of resources. Additionally, one type of hardware resource here might be relative to multiple uses. For example, SRAM can also be used to implement hash-based exact matching. All these make the resource allocation on RMT-like pipeline complex but worth investigation.

Disruption-free runtime. Additionally, Menshen's current approach ensures isolation but ignores the inefficiency in terms of that it might be possible to dynamically adjust the amounts of resources allocated to different programs. For example, different measurement programs [71] might need different amounts of resources to fulfill their goals (*e.g.,* finding heavy hitters), according to the traffic they are experiencing [142]. This means that it will increase the efficiency of resource usage if we can support resource reallocation dynamically at runtime. To achieve that, two major aspects should be considered: how to safely and disruption-freely carry out the reallocation decision.

Memory access in RMT. Current design of RMT—as introduced in Chapter 2—scatters on-device stateful memory across different stages. With Menshen's isolation mechanisms, it still needs careful management to access the stateful memories. How to provide unified access to those on-device memory still requires new designs.

Performance isolation. Both Menshen and QingNiao do not deal with isolating traffic from different programs competing for the link bandwidth, which is a traffic management problem. Proposals like PIFO [128] can be used here, by assigning PIFO ranks to different programs to realize a desired bandwidth-sharing policy.

Load balancing in QingNiao. Policy-wise, QingNiao currently only supports pre-configured static rules rather than dynamically adjusting the rules according to the live traffic. To achieve that, we can extend QingNiao by monitoring traffic load in the QingNiao controller and periodically updating dispatch rules on hardware.

Other L7 processing. L7 dispatch does not involve any modification to the original L7 message. However, other L7 processing (*e.g.,* compression, serialization, etc.) may alter the length of the original L7 message. Purely packet-processing pipeline (*e.g.,* RMT or QingNiao) can not be directly applied to handling this type of L7 processing. Such pipelines may not be an ideal architecture to carry out this type of L7 processing. One potential solution may integrate pipeline processing with general cores like Pensando SmartNICs [2]: pipeline processing is used to dispatch the L7 messages to different cores for the modifications.

Bibliography

[1] About Arm Debugger Support for Overlays. https://developer.arm.com/documentation/101470/2021-0/Debugging-Embedded-Systems/About-Arm-Debugger-support-for-overlays?lang=en.

[2] AMD Pensando Networking. https://www.amd.com/en/products/accelerators/pensando.html.

[3] AMD Xilinx Alveo U250 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html.

[4] Before Memory was Virtual. http://160592857366.free.fr/joe/ebooks/ShareData/Before%20Memory%20was%20Virtual%20By%20Peter%20J.%20Denning%20from%20George%20Mason%20University.pdf.

[5] Behavioral model targets. https://github.com/p4lang/behavioral-model/blob/master/targets/README.md.

[6] Better Load Balancing: Real-Time Dynamic Subsetting. https://www.uber.com/blog/better-load-balancing-real-time-dynamic-subsetting/.

[7] Bidirectional Forwarding Detection (BFD). https://tools.ietf.org/html/rfc5880.

[8] Cisco Nexus SmartNIC. https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html.

[9] CloudLab. https://cloudlab.us/.

[10] Corundum Github Repository. https://github.com/corundum/corundum/commits/56fe10f27d9b42f1ff9abe4d735b113008e4be9d.

[11] Daisy chain. https://en.wikipedia.org/wiki/Daisy_chain_(electrical_engineering).

[12] Data Plane Development Kit. https://www.dpdk.org/.

[13] DOJO: Super-Compute System Scaling for ML Training. https://hc34.hotchips.org/assets/program/conference/day2/Machine%20Learning/Hotchip%20Dojo%20System%20v25.pdf.

[14] Envoy Proxy. https://www.envoyproxy.io/.

[15] Examples of funclatency, the eBPF/BCC version. https://github.com/iovisor/bcc/blob/master/tools/funclatency_example.txt.

[16] Fast HTTP Package for Go. https://github.com/valyala/fasthttp.

[17] FreePDK45. https://www.eda.ncsu.edu/wiki/FreePDK45:Contents.

[18] Github Repository of RingLeader. https://github.com/utnslab/RingleaderNIC.

[19] Github Repository of RocksDB. https://github.com/facebook/rocksdb.

[20] Google's PSP cryptographic hardware offload at scale is now open source. https://cloud.google.com/blog/products/identity-security/announcing-psp-security-protocol-is-now-open-source.

[21] Intel Mount Evans IPU. https://www.intel.com/content/www/us/en/products/details/network-io/ipu/e2000-asic.html.

[22] Intel Tofino Series Programmable Ethernet Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[23] Introduction to gRPC. https://grpc.io/docs/what-is-grpc/introduction/.

[24] Istio Service Mesh. https://istio.io/.

[25] Leveraging Stratum and Tofino Fast Refresh for Software Upgrades. https://opennetworking.org/wp-content/uploads/2018/12/Tofino_Fast_Refresh.pdf.

[26] Linkerd: The world's most advanced service mesh. https://linkerd.io/.

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

[28] Marvell LiquidIO III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf.

[29] Mellanox BlueField VPI 100Gps SmartNIC. https://www.mellanox.com/files/doc-2020/pb-bluefield-vpi-smart-nic.pdf.

[30] Mellanox Innova Open Programmable SmartNIC. https://www.mellanox.com/sites/default/files/doc-2020/pb-innova-2-flex.pdf.

[31] Microservices on AWS. https://aws.amazon.com/microservices/.

[32] Netflix Ribbon. https://github.com/Netflix/ribbon.

[33] NetFPGA-SUME Virtex-7 FPGA Development Board. https://reference.digilentinc.com/reference/programmable-logic/netfpga-sume/start.

[34] NGINX HTTP Load Balancing. https://docs.nginx.com/nginx/admin-guide/load-balancer/http-load-balancer/.

[35] NVIDIA Data Processing Units. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[36] NVIDIA Mellanox ConnectX-5 Adapters. https://www.nvidia.com/en-us/networking/ethernet/connectx-5/.

[37] Operating Systems Three Easy Pieces. https://iitd-plos.github.io/os/2020/ref/os-arpaci-dessau-book.pdf.

[38] Overlaying in Commodore. https://www.atarimagazines.com/compute/issue73/loading_and_linking.php.

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

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

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

[42] Programmable Forwarding Planes are Here to Stay. https://conferences.sigcomm.org/sigcomm/2017/files/program-netpl/01-mckeown.pptx.

[43] Protobuf. https://protobuf.dev/.

[44] Receive Side Scaling. https://www.kernel.org/doc/Documentation/networking/scaling.txt.

[45] Regular Expression. https://en.wikipedia.org/wiki/Regular_expression.

[46] [RFC 7540] Hypertext Transfer Protocol Version 2 (HTTP/2). https://www.rfc-editor.org/rfc/rfc7540.

[47] [RFC 793] Transmission Control Protocol. https://www.ietf.org/rfc/rfc793.txt.

[48] [RFC 9000] QUIC: A UDP-Based Multiplexed and Secure Transport. https://www.rfc-editor.org/rfc/rfc9000.html.

[49] Spirent Quint-Speed High-Speed Ethernet Test Modules. https://assets.ctfassets.net/wcxs9ap8i19s/12bhgz12JBkRa66QUG4N0L/af328986e22b1694b95b290c93ef6c21/Spirent_fX3_HSE_Module_datasheet.pdf.

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

[51] Synapse: A transparent service discovery framework for connecting an SOA . https://airbnb.io/projects/synapse/.

[52] Synopsys DC Ultra. https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html.

[53] Tesla's TTPoE: Replacing TCP for Low Latency Applications. https://chipsandcheese.com/2024/08/27/teslas-ttpoe-at-hot-chips-2024-replacing-tcp-for-low-latency-applications/.

[54] The Space Shuttle Fight Software Development Process. https://www.nap.edu/read/2222/chapter/5.

[55] Transport Layer Security. https://en.wikipedia.org/wiki/Transport_Layer_Security.

[56] Type-Length-Value (TLV) Encoding. https://en.wikipedia.org/wiki/TypeâĂŞlengthâĂŞvalue.

[57] Uniform Resource Locator (URL). https://en.wikipedia.org/wiki/URL.

[58] Virtual Function I/O. https://docs.kernel.org/driver-api/vfio.html.

[59] Von Neumann Architecture. https://en.wikipedia.org/wiki/Von_Neumann_architecture.

[60] wrk - A Modern HTTP Benchmarking Tool. https://github.com/wg/wrk.

[61] Xilinx AXI4-Lite Interface Protocol. https://www.xilinx.com/products/intellectual-property/axi.html.

[62] Xilinx AXI4-Stream. https://www.xilinx.com/products/intellectual-property/axi4-stream_interconnect.html.

[63] Xilinx Block Memory Generator v8.4. https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_4/pg058-blk-mem-gen.pdf.

[64] Xilinx Parameterizable Content-Addressable Memory. https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf.

[65] Xilinx UltraScale Architecture Memory Resources User Guide (UG573). https://docs.xilinx.com/v/u/en-US/ug573-ultrascale-memory-resources.

[66] Xilinx Vitis High-Level Synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[67] M. B. Anwer, M. Motiwala, M. b. Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. In *ACM SIGCOMM*, 2010.

[68] K. Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. *IEEE TC*, 1990.

[69] A. Bakre and B. Badrinath. I-TCP: Indirect TCP for Mobile Hosts. In *IEEE ICDCS*, 1995.

[70] M. Becchi and P. Crowley. A Hybrid Finite Automaton for Practical Deep Packet Inspection. In *ACM CoNEXT*, 2007.

[71] R. Ben Basat, S. Ramanathan, Y. Li, G. Antichi, M. Yu, and M. Mitzenmacher. PINT: Probabilistic In-band Network Telemetry. In *ACM SIGCOMM*, 2020.

[72] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight Remote Procedure Call. *ACM TOCS*, 1990.

[73] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN. In *ACM SIGCOMM*, 2013.

[74] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hXDP: Efficient Software Packet Processing on FPGA NICs. In *USENIX OSDI*, 2020.

[75] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall. dRMT: Disaggregated Programmable Switching. In *ACM SIGCOMM*, 2017. Tech report available at https://cs.nyu.edu/~anirudh/sigcomm17_drmt_extended.pdf.

[76] T. Cui, C. Zhao, W. Zhang, K. Zhang, and A. Krishnamurthy. Laconic: Streamlined Load Balancers for SmartNICs, 2024.

[77] H. M. Demoulin, J. Fried, I. Pedisich, M. Kogias, B. T. Loo, L. T. X. Phan, and I. Zhang. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In *ACM SOSP*, 2021.

[78] J. B. Dennis and D. P. Misunas. A Preliminary Architecture for a Basic Data-Flow Processor. In *ACM/IEEE ISCA*, 1974.

[79] S. Dharanipragada, S. Joyner, M. Burke, J. Nelson, I. Zhang, and D. R. K. Ports. PRISM: Rethinking the RDMA Interface for Distributed Systems. In *ACM SOSP*, 2021.

[80] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM IMC*, 2015.

[81] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *USENIX ATC*, 2019.

[82] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, H. K. Chandrappa, S. Chaturmohta, M. Humphrey, J. Lavier, N. Lam, F. Liu, K. Ovtcharov, J. Padhye, G. Popuri, S. Raindel, T. Sapre, M. Shaw, G. Silva, M. Sivakumar, N. Srivastava, A. Verma, Q. Zuhair, D. Bansal, D. Burger, K. Vaid, D. A. Maltz, and A. Greenberg. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX NSDI*, 2018.

[83] A. Forencich, A. C. Snoeren, G. Porter, and G. Papen. Corundum: An Open-Source 100-Gbps NIC. In *IEEE FCCM*, 2020.

[84] J. Fried, G. I. Chaudhry, E. Saurez, E. Choukse, I. Goiri, S. Elnikety, R. Fonseca, and A. Belay. Making Kernel Bypass Practical for the Cloud with Junction. In *USENIX NSDI*, 2024.

[85] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay. Caladan: Mitigating Interference at Microsecond Timescales. In *USENIX OSDI*, 2020.

[86] R. Gandhi, Y. C. Hu, and M. Zhang. Yoda: A Highly Available Layer-7 Load Balancer. In *ACM EuroSys*, 2016.

[87] J. Gao, E. Zhai, H. H. Liu, R. Miao, Y. Zhou, B. Tian, C. Sun, D. Cai, M. Zhang, and M. Yu. Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In *ACM SIGCOMM*, 2020.

[88] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[89] G. Gibb, G. Varghese, M. Horowitz, and N. McKeown. Design Principles for Packet Parsers. In *ACM/IEEE ANCS*, 2013.

[90] V. Gogte, A. Kolli, M. J. Cafarella, L. D'Antoni, and T. F. Wenisch. HARE: Hardware Accelerator for Regular Expressions. In *IEEE MICRO*, 2016.

[91] S. Grant, A. Yelam, M. Bland, and A. C. Snoeren. SmartNIC Performance Isolation with FairNIC: Programmable Networking for the Cloud. In *ACM SIGCOMM*, 2020.

[92] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM*, 2009.

[93] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. RDMA over Commodity Ethernet at Scale. In *ACM SIGCOMM*, 2016.

[94] D. Hancock and J. van der Merwe. HyPer4: Using P4 to Virtualize the Programmable Data Plane. In *ACM CoNEXT*, 2016.

[95] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker. Modular Switch Programming Under Resource Constraints. In *USENIX NSDI*, 2022.

[96] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In *ACM/SIGDA FPGA*, 2019.

[97] J. Jang, S. J. Jung, S. Jeong, J. Heo, H. Shin, T. J. Ham, and J. W. Lee. A Specialized Architecture for Object Serialization with Applications to Big Data Analytics. In *ACM/IEEE ISCA*, 2020.

[98] W. Jiang. Scalable Ternary Content Addressable Memory Implementation Using FPGAs. In *ACM/IEEE ANCS*, 2013.

[99] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica. NetChain: Scale-Free Sub-RTT Coordination. In *USENIX NSDI*, 2018.

[100] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *ACM SOSP*, 2017.

[101] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *USENIX NSDI*, 2015.

[102] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *USENIX NSDI*, 2019.

[103] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks. Profiling a Warehouse-Scale Computer. In *ACM/IEEE ISCA*, 2015.

[104] S. Karandikar, C. Leary, C. Kennelly, J. Zhao, D. Parimi, B. Nikolic, K. Asanovic, and P. Ranganathan. A Hardware Accelerator for Protocol Buffers. In *IEEE MICRO*, 2021.

[105] J. Krude, J. Hofmann, M. Eichholz, K. Wehrle, A. Koch, and M. Mezini. Online Reprogrammable Multi Tenant Switches. In *1st ACM CoNEXT Workshop on Emerging In-Network Computing Paradigms*, 2019.

[106] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen. ClickNP: Highly Flexible and High Performance Network Processing with Reconfigurable Hardware. In *ACM SIGCOMM*, 2016.

[107] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX OSDI*, 2016.

[108] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh, and M. Yu. HPCC: High Precision Congestion Control. In *ACM SIGCOMM*, 2019.

[109] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *USENIX NSDI*, 2014.

[110] K. Lim, M. Giordano, T. Stavrinos, I. Zhang, J. Nelson, B. Kasikci, and T. Anderson. Beehive: A Flexible Network Stack for Direct-Attached Accelerators. In *IEEE MICRO*, 2024.

[111] J. Lin, A. Cardoza, T. Khan, Y. Ro, B. E. Stephens, H. Wassel, and A. Akella. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *USENIX NSDI*, 2023.

[112] J. Lin, K. Patel, B. E. Stephens, A. Sivaraman, and A. Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *USENIX OSDI*, 2020.

[113] P. Linz and S. H. Rodger. *An introduction to formal languages and automata*. Jones & Bartlett Learning, 2022.

[114] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading Distributed Applications onto SmartNICs Using IPipe. In *ACM SIGCOMM*, 2019.

[115] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities. In *ACM SIGCOMM*, 2018.

[116] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX NSDI*, 2019.

[117] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 2004.

[118] Y. Park, H. Park, and S. Mahlke. CGRA Express: Accelerating Execution Using Dynamic Operation Fusion. In *ACM CASES*, 2009.

[119] B. Pismenny, H. Eran, A. Yehezkel, L. Liss, A. Morrison, and D. Tsafrir. Autonomous NIC Offloads. In *ACM ASPLOS*, 2021.

[120] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda, F. Huici, and G. Siracusano. FlowBlaze: Stateful Packet Processing in Hardware. In *USENIX NSDI*, 2019.

[121] A. Pourhabibi, S. Gupta, H. Kassir, M. Sutherland, Z. Tian, M. P. Drumond, B. Falsafi, and C. Koch. Optimus Prime: Accelerating Data Transformation in Servers. In *ACM ASPLOS*, 2020.

[122] A. Pourhabibi, M. Sutherland, A. Daglis, and B. Falsafi. Cerebros: Evading the RPC Tax in Datacenters. In *IEEE MICRO*, 2021.

[123] S. Qi, L. Monis, Z. Zeng, I.-c. Wang, and K. K. Ramakrishnan. SPRIGHT: Extracting the Server from Serverless Computing! High-Performance EBPF-Based Event-Driven, Shared-Memory Processing. In *ACM SIGCOMM*, 2022.

[124] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo. Limago: An FPGA-Based Open-Source 100 GbE TCP/IP Stack. In *IEEE FPL*, 2019.

[125] H. Saokar, S. Demetriou, N. Magerko, M. Kontorovich, J. Kirstein, M. Leibold, D. Skarlatos, H. Khandelwal, and C. Tang. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *USENIX OSDI*, 2023.

[126] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. K. Ports, and P. Richtarik. Scaling Distributed Machine Learning with In-Network Aggregation. In *USENIX NSDI*, 2021.

[127] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja. Hard Virtualization of P4-Based Switches with VirtP4. In *ACM SIGCOMM Posters and Demos*, 2019.

[128] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable Packet Scheduling at Line Rate. In *ACM SIGCOMM*, 2016.

[129] H. Soni, M. Rifai, P. Kumar, R. Doenges, and N. Foster. Composing Dataplane Programs with μP4. In *ACM SIGCOMM*, 2020.

[130] B. E. Stephens, D. Grassi, H. Almasi, T. Ji, B. Vamanan, and A. Akella. TCP is Harmful to In-Network Computing: Designing a Message Transport Protocol (MTP). In *ACM HotNets*, 2021.

[131] R. Stoyanov and N. Zilberman. MTPSA: Multi-Tenant Programmable Switches. In *3rd P4 Workshop in Europe*, 2020.

[132] M. Sutherland, S. Gupta, B. Falsafi, V. Marathe, D. Pnevmatikatos, and A. Daglis. The NEBULA RPC-Optimized Architecture. In *ACM/IEEE ISCA*, 2020.

[133] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon. P4FPGA: A Rapid Prototyping Framework for P4. In *ACM SOSR*, 2017.

[134] T. Wang, J. Lin, G. Antichi, A. Panda, and A. Sivaraman. Application-Defined Receive Side Dispatching on the NIC, 2024.

[135] T. Wang, X. Yang, G. Antichi, A. Sivaraman, and A. Panda. Isolation Mechanisms for High-Speed Packet-Processing Pipelines. In *USENIX NSDI*, 2022.

[136] T. Wang, H. Zhu, F. Ruffy, X. Jin, A. Sivaraman, D. R. K. Ports, and A. Panda. Multitenancy for Fast and Programmable Networks in the Cloud. In *USENIX HotCloud*, 2020.

[137] J. Xing, K.-F. Hsu, M. Kadosh, A. Lo, Y. Piasetzky, A. Krishnamurthy, and A. Chen. Runtime Programmable Switches. In *USENIX NSDI*, 2022.

[138] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu. HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane. In *IEEE ICCCN*, 2017.

[139] P. Zheng, T. Benson, and C. Hu. P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs. In *ACM CoNEXT*, 2018.

[140] P. Zheng, T. Benson, and C. Hu. ShadowP4: Building and Testing Modular Programs. In *ACM SIGCOMM Posters and Demos*, 2018.

[141] H. Zhu, K. Kaffes, Z. Chen, Z. Liu, C. Kozyrakis, I. Stoica, and X. Jin. RackSched: A Microsecond-Scale Scheduler for Rack-Scale Computers. In *USENIX OSDI*, 2020.

[142] H. Zhu, T. Wang, Y. Hong, D. R. K. Ports, A. Sivaraman, and X. Jin. NetVRM: Virtual Register Memory for Programmable Networks. In *USENIX NSDI*, 2022.

[143] X. Zhu, G. She, B. Xue, Y. Zhang, Y. Zhang, X. K. Zou, X. Duan, P. He, A. Krishnamurthy, M. Lentz, D. Zhuo, and R. Mahajan. Dissecting Overheads of Service Mesh Sidecars. In *ACM SoCC*, 2023.

[144] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, 2014.