Rules Offload Engine (ROE): Accelerating Host SDN Policy Evaluation

Anshuman Verma, Tian Tan, Ahmed Abdelsalam, Milan Dasgupta, Jonathan Hunter, Zach Libby, Narayanan Ravichandran, Harish Srinivasan, Lok Chand Koppaka, Matt Reat, Nadeen Gebara, Vishal Gondaliya, Ezz Hamed, Rahul Garlapati, Abdullah Mughrabi, Dev Desai, Alexander Malysh, Shwetha Bhat, Rohan Kandi, Megan Sng, Tushar Garg, Muluken Hailesellasie, Andrew Putnam, Derek Chiou, Osman Ertugay, Alireza Dabagh, Vivek Bhanu, Daniel Firestone*
Microsoft
Redmond, WA, USA

Abstract

Software Defined Networking (SDN) policy evaluation analyzes and verifies network rules to enable accurate packet routing, manage access controls, and ensure security and privacy. Therefore, SDN policy evaluation is a mandatory step that each network packet must go through in virtual machines hosted on public cloud services, such as Azure. Evaluating SDN policies requires up to several hundred microseconds, which severely limits performance. The problem exacerbates at cloud scale where more *connections per second (CPS)* are desirable across millions of virtual machines.

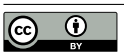
In this paper, we present *Rules Offload Engine (ROE)*, a hardware-software co-designed solution that accelerates SDN policy evaluation through FPGA-based SmartNICs. We propose the *ROE Instruction Set Architecture (RISA)* that features dedicated networking-friendly instructions. RISA fuses control policy data and operation directly into the instructions to facilitate an efficient FPGA implementation. We develop an *ROE compiler* that translates SDN rules into RISA instructions. We also design the *ROE-Core* that implements RISA and other components for seamless integration with existing hardware and software stacks. Overall, ROE enables up to 400K CPS, achieving 10× higher throughput compared to prior software-centric approaches. ROE is deployed in Azure and was featured as a part of the second-generation Azure Boost.

CCS Concepts

• **Hardware → Hardware accelerators**; **Reconfigurable logic applications**; • **Computer systems organization → Reconfigurable computing**; • **Networks → Network Adapters**.

ACM Reference Format:
Anshuman Verma, Tian Tan, Ahmed Abdelsalam, Milan Dasgupta, Jonathan Hunter, Zach Libby, Narayanan Ravichandran, Harish Srinivasan, Lok Chand Koppaka, Matt Reat, Nadeen Gebara, Vishal Gondaliya, Ezz Hamed, Rahul Garlapati, Abdullah Mughrabi, Dev Desai, Alexander Malysh, Shwetha Bhat, Rohan Kandi, Megan Sng, Tushar Garg, Muluken Hailesellasie, Andrew Putnam, Derek Chiou, Osman Ertugay, Alireza Dabagh, Vivek Bhanu, Daniel Firestone. 2026. Rules Offload Engine (ROE): Accelerating Host SDN Policy Evaluation. In *ACM SIGCOMM 2026 Conference (SIGCOMM '26), August 17–21, 2026, Denver, CO, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3789240.3829118

*The corresponding authors can be reached at ansverma@microsoft.com, naravich@microsoft.com, zachl@microsoft.com, and harishs@microsoft.com. Ezz Hamed and Tushar Garg contributed to the project while they were at Microsoft.



SIGCOMM '26, Denver, CO, USA

ACM ISBN 979-8-4007-2467-1/2026/08
https://doi.org/10.1145/3789240.3829118

1 Introduction

Cloud service providers rely on *Software Defined Networking (SDN)* policies to implement complex network architectures and security semantics in multi-tenant environments [2, 16, 20]. These semantics include layered access control, security group enforcement, network address translation, tunneling, load balancing, and virtual network isolation. SDN policies are reviewed by the cloud provider for *every single packet* at the scale of millions of connections across thousands of endpoints. Evaluating SDN policies for a packet is computationally intensive and can take up to thousands of host CPU cycles, incurring latency overheads in the order of hundreds of microseconds [13]. Reducing the overheads of SDN policy evaluation is therefore critical to improve performance.

Microsoft's Virtual Filtering Platform (VFP) [13] and AccelNet [14] are key technologies empowering Azure to reduce networking overheads and are currently deployed at scale. To establish new connections, Azure relies on the software-based VFP to process multiple SDN layers sequentially using Azure policy specific rules. This computationally intensive process produces a connection-specific packet transposition. Once a connection has been established, VFP reuses the computational results from the SDN policy evaluation steps by storing them in its internal data structures to process subsequent packets as long as the policies themselves remain unchanged since connection establishment. AccelNet further accelerates this step for already established connections by caching the SDN policy evaluation results and looking them up from tables in specialized hardware [14], as shown in Figure 1(a). However, AccelNet cannot speed up new connection setups because it simply does not have the cached SDN policy evaluation results. For these *exception packets* that AccelNet cannot process, new connection establishments must still rely on software-based VFP processing, which is slow and limits performance when individual connections are short-lived and aggregate connection setup rates are very high. This acts as a critical barrier to scaling cloud networking performance. In this paper, we present *Rules Offload Engine (ROE)*, an accelerator that successfully bridges this gap and is currently deployed in Azure.

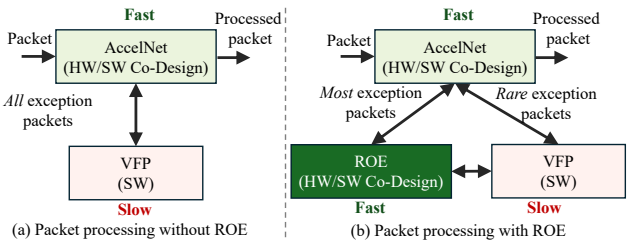


Figure 1: (a) AccelNet [14] quickly processes packets for previously established connections by looking up SDN rules. *All* exception packets corresponding to new connection setups are processed by Virtual Filtering Platform (VFP) [13] by traversing SDN layers for the packet using Azure policy specific rules. The SDN metadata is cached in AccelNet for subsequent packets. Slow VFP processing in software limits new connection establishment rates. Rules Offload Engine (ROE) overcomes this issue by processing *most* exception packets in hardware, while VFP only processes *rare* ones.

SDN policies are evaluated per port or virtual network interconnect (vNIC) on a virtual machine (VM). A VM can have multiple vNICs, each governed by its own SDN policy. Each vNIC policy further is composed of layers that may filter and modify packets incrementally and rules that define matching conditions. Layer traversal is sequential and rules are usually organized in pointer-based data structures, like trees, making software-based approaches too slow because they heavily rely on priority resolution and condition-based branching in rule evaluation and metadata propagation between layers. These execution models map poorly onto datapath pipelines optimized for uniform stateless packet processing as they cannot efficiently handle conditional branching, state information, and metadata propagation [21]. Programmable datapaths, like P4 targets [31], are constrained by fixed pipeline depth, limited control flow, and inability to efficiently express priority-ordered, conditionally branch based rule evaluation. In contrast, embedded CPU cores incur instruction fetch, memory accesses, and synchronization overheads [29]. ROE overcomes these drawbacks using a hardware-software co-design approach that features network-friendly instructions to reduce computational costs and an efficient architecture for accelerated rule matching and layer traversal.

ROE features the (1) *ROE Instruction Set Architecture (RISA)*, (2) RO-Bridge for interfacing, the RISA-compiler that maps SDN policies into RISA programs and RO-Bridge to interface with VFP and AccelNet, (3) an ROE-Core that employs micro-architectural optimizations for efficient FPGA implementations, and (4) other necessary hardware/software interfaces and micro-engines for seamless deployment in Azure. RISA features dedicated read-only instructions, eliminating frequent store operations. Note that stores may still be required occasionally for state management but they exploit the pre-existing, heavily optimized, AccelNet infrastructure. RISA includes networking optimized instructions for rule matching operations that enables ROE to speed up policy evaluations. RISA also supports data and control fusion for quicker processing and efficient FPGA implementation by reducing the area overheads through removal of data-caches and load-store units.

We build the RO-bridge, a software control plane, that orchestrates packet processing by AccelNet, ROE, and VFP. We develop the RISA-Compiler, a just-in-time compiler that receives vNIC SDN polices as an input and generates a micro-architecture aware highly-efficient RISA program. The compiler also supports seamless SDN policy updates by generating a new variant of the executable to accommodate the update and mapping it to a new page on the CPU. Then, it inserts a fence and processes any remaining packets using the older SDN policy executable. Once complete, it flushes the ROE-Core caches, migrates to the new page, and discards the old page which is re-purposed by the software later.

We design the ROE-Core, a six-core engine, where each core employs an 8-stage pipeline, supports eight threads, and incorporates delayed branching. These micro-architectural optimizations allow faster computations and efficient handling of branches [24]. The ROE-Core also includes a networking data structure aware register file to store both the original and modified packet contents for SDN policy evaluation. The register file is statically partitioned for each thread, which maps to a new connection request. Each core also has a private L1 instruction cache and all cores have a shared L2 instruction cache. Finally, the ROE-Core supports breakpoints and other debugging features that are critical for deployments at scale.

ROE leverages existing AccelNet infrastructure and receives processed information, such as port-id, packet profile, parsed network tuples, and additional metadata rather than the packet itself. This accelerates SDN policy evaluation. Once a policy has been evaluated, it offloads a *unified match-action flow entry* to AccelNet using a custom micro-engine. This entry maps SDN policy evaluation into a combined action and gets cached in AccelNet for subsequent packet processing. This feature enables line-rate execution of packets once a connection is established. It also introduces additional hardware engines for connection management for removal of match-action flow entries after a flow expires or a connection teardown. Additionally, it supports live-migration and save-restore functionality to enable firmware roll-outs in the datacenter.

Overall, this paper makes the following contributions:

(1) We introduce a networking optimized *ROE Instruction Set Architecture (RISA)* that embeds packet semantics directly into instructions. This enables us to eliminate expensive data-caches and load-store units for efficient FPGA implementation.
(2) We build the RO-Bridge for interfacing with AccelNet and VFP and the RISA-Compiler that maps SDN policies including rules, priorities, and protocol layers to ROE's execution model while preserving control plane expressivity.
(3) We design the ROE-Core, a multi-core, multi-threaded engine for SDN policy evaluation, using read-only instructions to deliver high throughput and low latency connection establishments.
(4) We deploy an end-to-end hardware-software co-designed solution that evaluates SDN policies, supports connection life cycle management, and seamlessly interacts with existing accelerators on Azure to achieve line-rate performance.

ROE supports up to 400K connections per second on existing hardware via only an FPGA image rollout and is currently deployed at hyperscale in Azure.

2 Background

2.1 SDN Policy Evaluation

SDN policy evaluation forms the backbone of packet processing in cloud networks. A VM comprises one or more vNICs, each implementing its own SDN policy through layers. For example, a Network Address Translation (NAT) layer translates a virtual address to a direct address, whereas a Metering layer counts the total number of bytes sent or received based on some classifiers. To implement individual functionalities, each SDN layer evaluates a policy by checking if certain fields of a packet *match* one or more conditions and updates packet field(s) as an *action*. It then sends the processed packet to the next layer. We show this using Figure 2. For simplicity, we only show two layers (L_0 and L_1) and a packet with three fields (❶, ❷, and ❸). Layer L_0 checks if the second field (❷) falls within an acceptable range and matches the condition for its rule. If the condition is met, L_0 takes an appropriate action and sends the updated packet to L_1, which checks for its own matching rules (on ❶ and ❸). SDN policy evaluation is control-intensive with sequential computation involving irregular memory accesses and branches that fundamentally limit performance on general-purpose or embedded CPUs due to cache misses, long-latency memory accesses, and branch mispredictions. At cloud scale, with networks dealing with thousands of connections on average per VM and an increasing number of VMs per node, accelerating SDN policy evaluation is critical because each packet goes through this step.

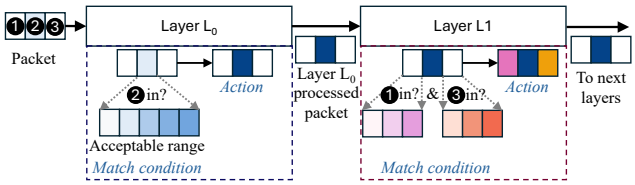


Figure 2: Illustration of SDN policy evaluation. Each SDN layer checks for its own matching rules and takes an action to process the packet before sending it to the next layer.

2.2 Virtual Filtering Platform (VFP)

VFP is at the heart of Azure's SDN policy enforcement running on millions of nodes. It performs the following three critical tasks.

1. Policy Evaluation: VFP evaluates SDN policies in Azure VMs using a priority-based multi-classifier search, such as trie, to identify an action (or *rule*) using a matching condition in an optimized data structure (for example, linked lists, hashmaps, interval trees). After completing policy evaluation, VFP creates a *unified flow (UF) entry* that coalesces every action performed on a packet in each SDN layer. It stores this UF entry in a software cache and uses it to process subsequent packets for the connection. VFP also features an advanced programming model that allows seamless policy modifications by the control plane and mechanisms to re-evaluate pre-established connections based on the policy updates.

2. Connection Management: A connection is bidirectional (send and receive paths) and each of these directions corresponds to a *flow*. VFP tracks the state of a TCP connection in an internal data structure by following the standard TCP state tracking algorithm []. To efficiently manage these data structures and keep the memory footprint in check when connections are created or torn down, VFP performs automated flow state tracking. For example, if a VM sends a TCP Reset packet, then VFP deletes its internal data structures for that connection and recovers the resources for future usage by following a specialized subroutine. Additionally, if a flow is idle for a stipulated time dictated by the SDN policy, VFP deletes this flow from its internal tracking structures after a period of inactivity and may optionally send a reset packet to both the server and the client. VFP also facilitates software load balancing by updating the routing information of a connection based on the redirect responses.

3. Quality of Service, Network Availability, and Monitoring: VFP enforces Quality of Service (QoS) by ensuring each VM strictly meets it connection-rate limits and its total number of flow quotas. It facilitates seamless live migration that allows VMs to continue using pre-established connections without requiring a re-handshake. VFP also ensures that its own software version updates can be performed seamlessly by efficiently saving its states (from the older version) and restoring these states in the new version's data structures within a bounded time with minimum packet retransmissions for VMs and without incurring any information loss. For monitoring purposes, it provides detailed flow logs per connection to customers for their internal usage, if requested. Additionally, VFP also exposes relevant telemetry at a vNIC granularity to support monitoring and operational diagnosis for deployment and diagnosis purposes.

2.3 AccelNet

Although VFP caches the match-action unified flow (UF) entry for each packet, retrieving this entry to process subsequent packets takes time. This is because the UF cache cannot be directly indexed using networking tuples as they can span up to a thousand bits. Instead, VFP organizes the UF cache using only a few tens of index bits computed using a hash function over the networking tuples. Thus, looking up the cache requires computing expensive hash functions. Moreover, VFP must also perform a full match between tuples used to create a hash and match fields of the UF to address any aliasing problem in hashing. Consequently, the UF cache lookup itself takes several cycles. AccelNet addresses this limitation with a match-action accelerator that caches the UF entries in a Generic Flow Table (GFT) and enables faster lookups by computing the index bits in hardware. Note that VFP still manages the installation and eviction of UF entries from GFT cache. Any packet is processed by querying the GFT for its corresponding UF entry. When a new connection is being set up, the entry cannot be found (GFT miss) and the processing request is sent to the VFP for this *exception packet*. Once the connection is fully established (for example, on handshake completion), VFP installs its corresponding UF entries in GFT. Subsequent packets hit in the GFT and AccelNet processes the packet using the GFT entry, completely bypassing VFP's software SDN stack, yielding massive performance improvements. However, such bypassing also prohibits VFP from observing packet activities for established connections and managing the state of the connection. To address this, AccelNet creates a copy of a few critical packets (such as TCP packet with RST or FIN flags) and sends it to VFP along with some metadata. VFP uses this copy-packet to

update its internal state and take appropriate action, like evicting or modifying a UF entry in GFT. To maintain accurate measurements for a packet and byte counts for a connection, AccelNet maintains per-flow counters that provide enriched data, such as timestamp for last activity, to VFP. Figure 3 gives an overview of packet processing with AccelNet and VFP (for more details, please see [13, 14]).

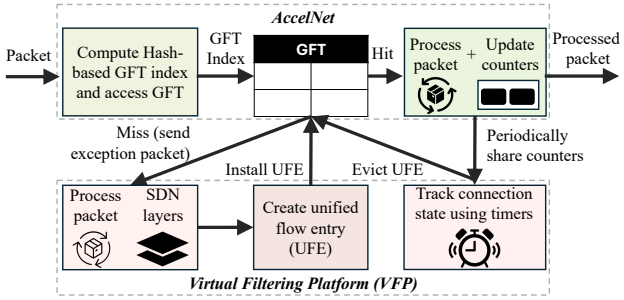


Figure 3: AccelNet bypasses VFP if a packet's UF entries are cached in GFT. Otherwise it sends the exception packet to VFP for SDN layer processing, after which VFP creates and installs the UF entries in GFT. AccelNet also maintains hardware counters and periodically shares them with VFP to track connection states and evict UF entries from GFT.

2.4 Challenges In Scaling Performance

While AccelNet enables fast packet processing once a connection has been set up, the first-time handling of a packet (exception packet) by software-based VFP on general-purpose or embedded CPU cores is slow due to many reasons. *First*, it involves irregular memory accesses, causing frequent cache misses and stalls due to long-latency memory accesses. *Second*, a large number of matching conditions transforms into proportionately large number of branches, subjecting the processing to frequent mis-prediction penalties. The problem worsens in public clouds due to different CPU generations and architectures, as VFP processing time is directly limited by the performance of these cores. *Third*, besides SDN policy evaluation time, offloading exception packets to VFP incurs I/O and queuing delays. VFP manages exception packets from multiple vNICs using shared queues, where each queue is processed by a specific core. In practice, the number of connection requests far exceeds the number of cores available. To ensure processing requests are balanced across cores, the Network Interconnect (NIC) selects a queue for each packet using Receive Side Scaling (RSS) hashing on packet tuples. This ensures that packets from a connection are processed by the same core to maintain in-order processing. But, hashing imperfections may still lead to load imbalances in the queues, causing delays. Also, VMs generate exception packets at different rates depending on the application. Queuing delays worsen when at least one of the VMs generates a disproportionately large number of exception packets. Consequently, software-based VFP processing severely limits connection establishment throughput. As many applications demand high connection-establishments rates and individual connections tend to be short-lived, this acts as a critical barrier to scaling cloud networking performance.

3 Rules Offload Engine (ROE)

This paper presents the *Rules Offload Engine (ROE)*, a networking optimized accelerator that enables fast SDN policy evaluation for new connections. It is an intermediate component between AccelNet and VFP. AccelNet sends the majority of the exception packets to ROE directly instead of VFP. ROE is not optimized for infrequent scenarios, such as address resolution protocol requests, to minimize hardware overheads. AccelNet sends these rare requests to VFP. ROE features a special-purpose networking-friendly ISA (RISA), RO-Bridge for interfacing with AccelNet and VFP, RISA-compiler to transform SDN policies into RISA instructions, FPGA optimized ROE-Core, and the necessary hardware-software interfaces for seamless integration into Azure's existing networking infrastructure. We discuss these in detail.

3.1 ROE Instruction Set Architecture (RISA)

SDN policy evaluation requires matching packets against rules across multiple groups spanning a sequence of layers. This process can be simplified as a series of rule traversals and conditional matching. Based on this insight, we design the *ROE Instruction Set Architecture (RISA)* to perform this step efficiently.

3.1.1 Registers. RISA supports both special purpose registers (SPRs) and general purpose registers (GPRs).

Special Purpose Registers (SPRs): ROE operates on packet headers. A packet header may include multiple encapsulations and a mandatory transport layer header. An encapsulation header or final transport header is called a header group. The maximum supported size of a header group is 128-bytes which is stored in the Header Group Register File (HGRF) with 32 registers of 32-bits each. Each register in the HGRF corresponds to a specific packet field or metadata, such as port-id, protocol types, IP and MAC addresses, etc. ROE supports SDN policy evaluations for up to two encapsulations. Header fields from these header groups reside in the HGRF. To store updated header groups, RISA offers 3 HGRFs per packet, resulting in 6 HGRFs in total. The HGRF is initialized by the ROE scheduler to packet header fields and metadata computed by AccelNet prior to the start of a thread. An HGRF register is accessed by setting a field in Header Control Register (HCR). To compute Sum-of-Products or Product-of-Sums, RISA offers the Main Condition Register (MCR), Scratch Condition Register (SCR), and Current Condition Register (CCR). To store intermediate results during hash computations, RISA includes the Hash Result Register (HRR).

General Purpose Registers (GPRs): RISA provides an additional 32 General Purpose Registers (GPRs) to store intermediate results.

3.1.2 Addressing Modes. SDN policy evaluation is primarily a read-only search operation that evaluates conditions and uses the computational results to update packet headers. RISA allows read-write access to packet header groups through HGR files and embedding the rules in the instructions themselves to eliminate the need to access operands using pointers. Connection states are updated internally in the hardware using AccelNet infrastructure. This eliminates the need for load-store operations and dedicated hardware units, making RISA amenable to efficient FPGA implementations. Hence, RISA only uses register and immediate addressing modes.

Table 1: Brief overview of RISA instructions. SR denotes source register operands, DR denotes destination register, imm denotes immediate values, OP denotes logical or arithmetic operation.

Instruction bit fields						Brief description
Bits[63:48]			Bits[47:32]		Bits[31:0]	
MATCH	SR1	SR2	XCR ctrl	shift or mask	imm	Operand comparison and computation of complex conditional expressions.
ALU	SR1	SR2	DR	OP	imm	Perform arithmetic or logical operation (OP) on SR1 and SR2, store result in DR
CTRL	SR1	OP	imm			Jump to a code section, call a sub-routine, or terminate program
HASH	SR1	mask	imm			Compute a 16-bit hash function using immediate key (imm)
μENGINE	ID	OP	Reserved			Offload complex functions to specialized hardware

3.1.3 Instructions. Each RISA instruction has a fixed length of 64-bits. RISA features the following types of instructions:
(1) **MATCH:** to facilitate most common operation in rule processing. MATCH instruction uses an XCR Control field to control how the MCR, SCR, and CCR are updated. For example, one can specify *MCR = SCR & CCR*. This capability significantly reduces the number of instructions required to evaluate Sum-of-Products (SOP) and Products-of-Sums (POS) expressions, which are common in SDN rule matching, thereby optimizing rule traversal efficiency. MATCH instructions natively support various comparison conditions (equal (EQ), greater than (GT), etc.), as well as mask and shift operations on operands. This allows frequent tasks, like prefix matching, to be expressed using a single instruction. Further, MATCH instructions support built-in conditional returns, enabling execution flow adjustment once a rule outcome is determined. This reduces the usage of control-flow instructions. For example, if a match finds **protocol=TCP** or **flags=SYN**, RISA allows sub-routines to immediately return to the program using just a single instruction, without requiring explicit conditional branches or program control instructions.

(2) **ALU:** bit-wise left and right shift (LSHFT, RSHFT) and logical operations (AND, OR, XOR) to support packet manipulations, addition and subtraction (ADD, SUB) to manage counter updates, and move instruction (MOV) to facilitate data movement from main memory to an ROE register or to set a specific value in the HCR.

(3) **HASH:** This supports efficient traversal of internal data structures, such as trie and interval-tree created by VFP and ROE-Compiler. The **HASH** instructions can leverage the mask field and HRR to chain hash results and compute 32-bit or 64-bit hash values with minimal computational overheads.

(4) μ**ENGINE:** As ROE leverages AccelNet's GFT, RISA supports these special instructions for interfacing with other micro-engines on the FPGA that are responsible for various operations. A key example is programming flows into GFT, which is handled through a dedicated instruction.

Overall, RISA relies on networking-optimized SPRs to reduce computational overheads, register and immediate addressing modes to lower memory accesses, specialized instructions tailored to match operations, and dedicated micro-engine operations to support seamless and efficient interfacing with AccelNet. Table 1 provides a brief description of the instruction types.

3.2 RO-Bridge and RISA-Compiler

The ROE software support features the (1) the RO-Bridge to manage the interactions of the ROE sub-system with VFP and AccelNet and (2) the RISA-Compiler to translate high-level SDN policies into RISA programs. Program generation begins when RO-Bridge retrieves policy state from VFP and forwards it to RISA-Compiler. RISA-Compiler then filters, sorts, and transforms the policy information into groups of classifiers optimized for execution on the underlying architecture. Figure 4 gives an overview of these components.

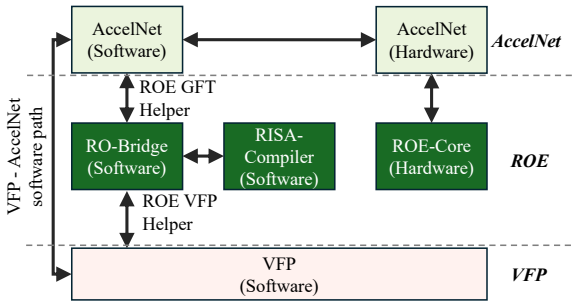


Figure 4: RO-Bridge facilitates interactions between AccelNet, ROE, and VFP.

3.2.1 RO-Bridge. It is the software control-plane and orchestration layer that translates VFP rule operations and coordinates with RISA-Compiler to generate RISA programs, manages ROE resources and its life-cycle, such as port creation and deletion, and maintains consistency between software and ROE rule processing. RO-Bridge enables seamless SDN policy modifications requested by VFP. For this, it requests the RISA-Compiler to regenerate the program for the updated policies, maps the compiled program to new code pages, requests a cache flush, and inserts a fence. ROE-Core flushes its caches and uses the fence to drain in-flight packets corresponding to older policies. Once the fence is completed, RO-Bridge allows the older code pages to be safely unmapped and freed for future use, ensuring a clean transition from older policies to newer ones. We discuss the micro-architectural support for this in Section 3.3.

3.2.2 RISA-Compiler Specifics and Optimizations. RISA-Compiler is responsible for converting SDN policy information, that originates from VFP, into a highly optimized RISA program.

RISA-Compiler generates the actual RISA program and writes instructions into *code pages*, each of which is a 64KB buffer allocated by the control-plane and backed by system memory that is directly accessible to the ROE hardware. A typical RISA program spans 10-15 such code pages. To avoid layout constraints imposed by page boundaries, RISA-Compiler inserts jump instructions at the end of each page, enabling logically contiguous execution regardless of the physical ordering of these pages. This provides many of the benefits of paged memory without requiring explicit runtime address management in RISA-Compiler. Once all instructions have been written into code pages, RO-Bridge enables ROE and programs the ROE hardware with the starting address in Program Counter (PC) of the port using AccelNet's helper interface.

Accelerated Policy Evaluation Through Rule Classification: VFP organizes SDN rules into classifiers to accelerate policy evaluation. These classifiers are rebuilt to adapt and optimize for ROE execution due to differences between VFP software and RISA execution models. RISA-Compiler constructs hardware micro-architecture-aware classifiers, including IP tries, interval trees, and hash tables (more details in Appendix **??**), and organizes them carefully to maximize cache hit rates. For example, IP trie nodes are aligned to ROE cache lines, so cache misses occur only when transitioning between nodes. This design enables large classifier sets to be translated into compact RISA programs with low and predictable latencies, allowing ROE to outperform VFP's software-based evaluation.

Speculative Reverse Flow: Network traffic is bidirectional with a forward flow from the client to the server that initiates SDN policy evaluation and a reverse flow from the server to client that may use these pre-computed evaluation results. ROE exploits this feature and introduces a speculative reverse-flow mechanism to lower reverse-direction processing costs [26]. During forward-flow creation, RISA-Compiler predicts and pre-offloads the reverse flow by swapping source and destination fields in HGR file and "undoing" any SDN policy applied in the modified HGR. If the prediction is correct, subsequent reverse packets are processed without additional RISA execution; otherwise, the speculative flow is discarded and the correct reverse flow is created upon receiving the reverse direction packets. To improve accuracy under encapsulation, RISA-Compiler leverages a *shadow* SDN policy layer containing rewritten outbound rules. This allows RISA-Compiler to infer expected encapsulation and apply them when constructing speculative reverse flows. In common workloads, this mechanism effectively doubles the connection establishment throughput while preserving correctness, and can be extended with extra shadow layers for other advanced and complex features.

Just-in-time Compilation For Dynamic Resource Management: ROE must support resources, such as IP address mapping, and SDN policy updates. A key challenge is that resource updates occur significantly more frequently than SDN policy updates. For example, in Azure, packet encapsulation and decapsulation frequently require translation between a virtual Customer Address (CA) and a physical Provider Address (PA) or the IP address mapping. Mappings are extremely dynamic because the number of VMs may scale up or down based on customer needs. In VFP, these CA/PA mappings are maintained in hash tables and updated at a high rate. However, in the presence of ROE, directly embedding mappings into RISA programs is impractical, as frequent updates would result in repeated ROE instruction cache flushes. To resolve this, RISA-Compiler treats mapping as a distinct resource with an independent compilation and update mechanism. Mappings are organized hierarchically: a global jump table maps each mapping space to a dedicated code page containing a hash table of mappings. The jump table is updated only during rule changes, while individual mapping updates require an atomic update to the corresponding jump instruction. The RO-Bridge batches mapping updates from VFP and triggers recompilation only when necessary. This just-in-time approach confines cache invalidation to mapping-specific code pages and allows ROE to sustain high mapping update rates without disrupting steady-state connection processing.

3.3 ROE-Core

To efficiently execute RISA instructions, we design the ROE-Core. We describe its micro-architecture next. ROE sub-system comprises six ROE-Cores, each employing the following optimizations.

Pipelined micro-architecture: ROE-Core features an eight-stage in-order pipeline that begins by fetching one instruction at a time in the *Fetch* stage. The instruction is then decoded and the HGRF and GPRs are accessed in the *Decode* stage. In the *Execute* stage, the instruction is dispatched to one of three execution units: MATCH, ALU, or HASH unit. Finally, the execution results are written back to the registers in the *Writeback* stage. Some stages (for instance, Fetch) take longer than others and are decomposed further to have a balanced implementation, resulting in a total of eight stages.

Fine-Grained Multi-Threading: To improve throughput and reduce stalls due to instruction dependencies, we use Fine-Grained Multi-Threading [24]. Each ROE-Core supports eight threads, matching the number of pipeline stages. Threads are scheduled in a round-robin fashion, with each thread fetching an instruction only once every eight cycles. This scheduling provides ample time for the results of a thread's previous instruction to arrive by the time its next instruction is scheduled. This minimizes dependency-based stalls in the pipeline to a large extent, but there may be rare cases when stalls may still occur, such as due to a cache miss. Due to massive connection-level parallelism in SDN workloads, ROE cores are able to maintain nearly 97% utilization under peak CPS workload.

Special Support For Control-Flow Instructions: To support CALL and RETURN instructions without relying on external memory, ROE-Core includes an on-chip stack. Control metadata, such as the next Program Counter (PC), is pushed onto the stack upon a CALL instruction and popped upon a RETURN instruction. As operations and data are fused in RISA instructions, the likelihood of encountering recursion is significantly reduced because recursive patterns commonly found in search algorithms are eventually unrolled into flattened instructions at compile time. This property eliminates the need for a deep stack, making an on-chip stack practical for our target workloads.

Multi-Port Register Files: Register files storing SPRs and GPRs support two read ports and one write port because each RISA instruction can only support up to two source register operands and one destination register. This enables single-cycle reads and writes. Register files (for both SPRs and GPRs) are maintained per thread.

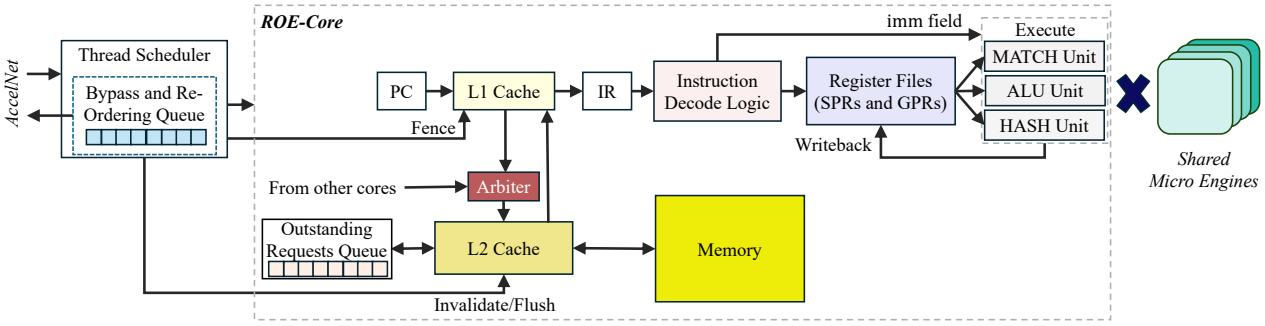


Figure 5: Overview of ROE-Core micro-architecture and its integration with AccelNet

Caches: ROE-Core consists of only instruction caches but no data caches. They are organized in a two-level read-only hierarchy with a line size of 256B (32 instructions). Each core has a private direct-mapped L1 cache of 32KB capacity statically partitioned uniformly across threads. Thus, each thread can only access 4KB. Static partitioning avoids cache thrashing in L1 caches. The cores also have access to a shared 256KB 4-way set associative L2 cache managed using LRU replacement policy. Table 2 provides an overview of the cache organization. The Fetch stage indexes the L1 cache using the PC of the active thread in that cycle and gets back the data within 2 cycles on a hit. On a miss, the request is sent to the L2 cache. A round-robin arbiter is used to send requests to the L2 cache. Its hit latency is 12 cycles, although more cycles may be needed due to arbitration. If an instruction misses in the L2 cache, a request is sent to the memory and tracked using the Outstanding Requests Queue (ORQ). The L2 cache is a non-blocking cache, allowing other requests to be serviced even in the presence of outstanding memory requests. Once the data is available, the line is installed in the L2 cache, the corresponding entry is serviced from the ORQ, and the data is sent to the respective core's L1 cache. This resolves the stall in the fetch stage for the corresponding thread, which now resumes.

Table 2: Overview of cache parameters

Parameter	L1 cache	L2 cache
Size	32 KB per core	256 KB for all cores
Associativity	Direct-Mapped	4-way set associative
Access	Private	Shared
Hit Latency	2 cycles	~12 cycles

Support For SDN Policy Updates: Recollect that ROE supports dynamic SDN policy modifications with the help of both hardware and software. ROE-Core invalidates cache lines from a specified address range or flushes all contents from the L2 cache depending on the modifications. Invalidation is preferred when SDN policy modifications are fairly small, involving only a few changed code pages (up to 500), such as in the case of rule updates. Cache flush is used otherwise. Fence operation flushes the contents of L1 cache and provides a completion to RO-Bridge when all of the active threads have successfully evicted contents of their respective cache regions. Figure 5 gives an overview of the ROE-Core micro-architecture.

3.4 Systems Support For ROE Integration

ROE leverages AccelNet infrastructure to create enriched metadata, such as packet profile, port-id, etc. that assists RISA-Compiler to generate more optimized RISA programs. This interfacing is done using dedicated components that are discussed next.

3.4.1 Micro-Engines. Micro-engines are RISA driven, special-purpose units optimized for certain operations that are not efficiently supported by ROE-Core. They are micro-programmed and invoked only when required, allowing us to optimize the latency of the ROE-Core pipeline for most common instructions. As micro-engines are used infrequently, they are shared between all cores. One such micro-engine used is the *Unified Flow Create Engine (UFCE)*. This engine is responsible for three primary tasks: 1) creation of a UFE based on the original and modified HGRFs used by the thread, 2) interacting with the GFT to add and update UFEs if needed, 3) error handling for a failed operation on a UFE by triggering appropriate actions, such as aborting, retrying, or raising an exception. There also exist other micro-engines, such as Packet Send Engine (PSE) that allows ROE to send a packet directly without using AccelNet, Debug Counter Engine (DCE) that provides enhanced logging.

3.4.2 Bypass and Re-ordering Queue. The Bypass and Re-ordering Queue (BRQ) preserves packet ordering [27]. Threads on ROE-Core run independently, which means packet processing is out-of-order. However, packets belonging to a connection must be processed in order to ensure networking semantics and consistent policy enforcement. Ideally, once the first packet is processed by ROE, AccelNet must process subsequent packets. However, a packet's UFE is not available in GFT when the first packet is being processed in ROE. In that case, AccelNet sends the subsequent packets to ROE. Redundant ROE processing is avoided by using the BRQ. Packets incoming to ROE are stored in the BRQ (in-order). A packet is dispatched to ROE-Core only if there is no other thread processing a packet for the same connection at that time. This is determined by using a hash function from AccelNet. Otherwise, the scheduler bypasses the packet from being dispatched to ROE-Core because a thread is already processing a different packet for the same connection. Packets are released to AccelNet from the top of BRQ (in-order) if the packet was dispatched to ROE-Core and its processing has completed or if the packet had bypassed and did not get scheduled to ROE-Core. This ensures that even if ROE-Core executes out-of-order, packets are still released in-order.

4 System Integration

Integrating ROE with AccelNet and VFP leads to three key challenges. *First*, how do we handle packets that cannot be processed by ROE? *Second*, how must ROE update already offloaded UFEs to AccelNet in the event of an SDN policy update, which could be communicated directly by VFP through RO-Bridge or realized indirectly through changes in packet's encapsulation tuples. *Third*, how do we ensure that ROE maintains feature compatibility with VFP for seamless integration into Azure's existing SDN infrastructure. Here, we discuss how we address these challenges.

4.1 VFP-Deferral

There are specific cases that ROE is not designed to handle in hardware because these scenarios are relatively uncommon and introducing hardware logic for them increases the complexity for FPGA-targeted implementations for little returns. An example of such a scenario is processing address resolution protocol requests. These packets bypass the ROE and AccelNet directly sends them to VFP for processing. We refer to this as the *VFP-deferral* process. Additionally, there may also be scenarios where a packet has arrived in ROE for processing but it cannot be processed due to exceptions, resource exhaustion, or failure to offload a UFE to AccelNet. ROE leverages VFP-deferral even for these cases but it is non-trivial.

The first challenge is safely cleaning up intermediate states created by ROE when a VFP-deferral occurs. ROE execution may encounter exceptions at any point during RISA program execution. For example, a RISA program may successfully create a forward flow but fail while processing the speculative reverse flow. In such cases, only one direction of the connection can be processed by AccelNet while the other direction must go to VFP, which makes state tracking of the full connection extremely cumbersome because of an awkward hardware software split. Hence, it is mandatory that both forward and reverse flows are managed by a single entity: ROE or VFP. To address this issue and avoid any flow duplication, ROE recovers any resources used in establishing the forward flow by sending this packet through AccelNet pipeline with a special request that recovers any state entries created during processing.

The second challenge is preserving deferral decisions. Once a connection has been deferred to VFP, its subsequent packets must continue to be reach VFP. This is ensured by tracking deferred packets (called blacklisted flows) and making AccelNet aware of them to ensure that subsequent packets are correctly routed to VFP. VFP-deferral also enables incremental system evolution, allowing a new features or SDN policy to be introduced in VFP first and later migrated to ROE as they become performance critical.

4.2 Re-evaluating Unified Flow Entries

SDN policies are dynamic and undergo frequent updates. After a policy update, any already established connection using the older polices must be re-evaluated using the newer ones. Moreover, during SDN policy evaluation, ROE-Compiler speculatively offloads a reverse flow that may also require an update. To address these cases, we implement a packet-triggered, flow re-evaluation mechanism invoked by AccelNet that allows ROE to review policy updates and modify any existing UFEs, as needed. This mechanism enables on-demand evaluation of active connections and avoids the need to update all UFEs en masse. For this, AccelNet utilizes a field already present in the UFE that indicates SDN policy version that created it and compares it against the current version of the SDN policy of a port. In the event of a mismatch, it invokes flow re-evaluation. Flow re-evaluation is also invoked in case a packet only matches the final transport header group of a UFE while the outer encapsulation header groups no longer match. In both cases, AccelNet sends a copy of the triggering packet and the associated state metadata to ROE. During flow re-evaluation, RoE may update the UFE and its state, or delete it entirely based on revised SDN policies.

4.3 Feature-Compatibility with VFP

AccelNet and VFP are already in deployment in Azure across millions of nodes. The incorporation of ROE must be done in a plug-and-play manner such that its inclusion remains almost invisible to the upper layers of the SDN control plane and VMs. It is also important that ROE remains functionally identical to VFP for any connection that it processes throughout its lifetime. We discuss the features that enable us to achieve these goals.

4.3.1 Connection Management. VFP removes any connections torn-down by peers and also periodically terminates any idle connections that cross a stipulated time-to-live (TTL) to reclaim resources and optionally sends TCP RST packets to relevant VMs. Additionally, it tracks state of TCP connection during a three-way handshake process. RoE achieves this feature by using a dedicated hardware, Flow Management Engine (FME), that periodically scans GFT and can delete UFEs on their TTL expiry, review its connection state and remove any torn-down connections [1]. This also requires us to augment AccelNet to maintain additional state information per TCP connection in GFT to track connection establishment and tear downs. This is feasible due to the reconfiguration ability of FPGAs. Additionally, FME supports accelerated removal of a vNIC by clearing its state, reading all the UFEs for a port to facilitate live-migration, or enabling enhanced monitoring by providing the logs for every UFE created for a vNIC.

4.3.2 Quality of Service (QoS). ROE features a hardware based QoS controller to restrict each VM to its allocated quota of connection establishment rate and UFE limits. The RO-Bridge programs per-VM flow limits and connections per second (CPS) budgets into the hardware. Once a VM exhausts either of its quota, then any new connections are dropped in the hardware. However, traffic on pre-existing flows remains unaffected.

4.3.3 Network Availability. Modern cloud networks require high availability and service continuity despite frequent software and FPGA updates. To keep customer network connections uninterrupted, ROE active flows are preserved to system memory by the DMA engine. After servicing is complete, the flows are restored by the DMA engine. This provides bounded and predictable performance impact, with brief, localized degradation during preservation and restoration while avoiding service interruption for co-resident VMs. For a fully populated GFT with approximately eight million ROE UFEs, both flow preservation and restoration complete within 600 ms each.

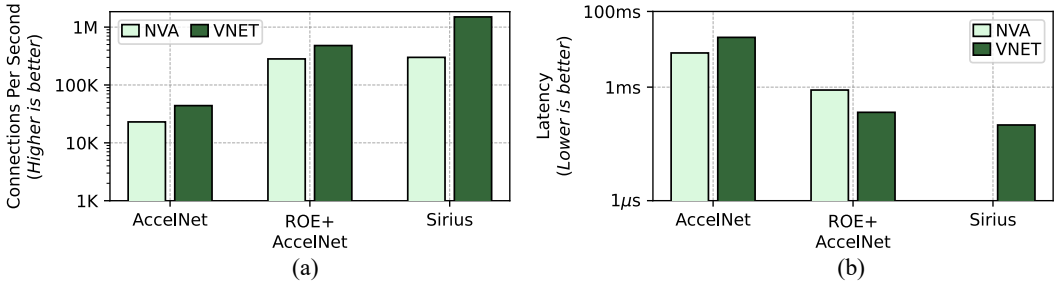


Figure 6: System-level performance comparison for different settings. ROE consistently improves throughput and latency.

4.3.4 Monitoring. ROE ensures visibility through comprehensive monitoring, particularly on packet drops and exception alerts. Warnings and errors are detected automatically, and recovery actions are triggered based on failure severity. Minor failures may be mitigated by terminating the faulty RISA program, while more severe faults may require functional or system-level resets. We further built dedicated tools for diagnostics, including a RISA emulator, which provides GDB-like instruction-level visibility of ROE core execution. In addition, we equip ROE with *Process Tuples*, which injects synthetic packets to compare hardware ROE execution against the RISA emulator. This tool enables fine-grained validation of SDN policy execution and flow decisions in ROE.

5 Evaluation Methodology

Baseline: We compare the system-level performance of ROE against a baseline that comprises AccelNet and VFP (corresponding to the most widely deployed SDN infrastructure in Azure prior to the inclusion of ROE). We also compare against the Sirius appliance prior work that uses a specialized ASIC that disaggregates network function processing off the host and into shared resource pools by making novel use of appliances [3].

Setup: For our evaluations, we deployed a pair of VMS on two production nodes with ROE+AccelNet (corresponding to our proposed approach) and AccelNet (corresponding to the baseline) are deployed on SmartNIC cards. One VM acts as a TCP client while the other one acts as a server, representing the most basic Virtual NETwork (VNET) scenario. We also evaluate a case with an optional middlebox for a Network Virtual Appliance (NVA) as a representative scenario that requires additional flow processing and state tracking. This setup was used for AccelNet baseline and ROE+AccelNet. For Sirius Appliance, we directly obtained its results from prior work. We have reviewed the evaluation of Sirius and confirmed that the experimental setup is equivalent. Thus, the results are comparable.

Figure-of-Merit: We evaluate the efficacy of ROE by measuring the established ***Connections Per Second (CPS)***. As ROE is primarily designed to accelerate connection establishment throughput, CPS is the most appropriate metric. We also compute the ***Latency*** of setting up a new connection by measuring the TCP SYN-to-SYNACK Round Trip Time (RTT). We leverage the NCPS tool [12] to stress test connection establishment and measure CPS and RTT. For Sirius Appliance, we directly obtain the results from the paper.

6 Experimental Results

In this section, we discuss the evaluation performance of ROE.

6.1 System-Level Performance At Scale

Figure 6 shows that ROE+AccelNet outperforms the baseline (AccelNet alone) and achieves up to 11× and 12.3× higher CPS on NVA and VNET setups respectively. Connection setups take up to an order of magnitude lower latencies on the NVA and VNET setups respectively. We observe that while the NVA setup has lower CPS than VNET with ROE, the connection setup latency is higher. Our investigation attributed this delay to the middlebox and additional network path to it because the processing time on the SmartNIC remains largely similar for both. Sirius outperforms ROE+AccelNet mainly due to two reasons. First, there exists SDN policy expression differences between Sirius and VFP. Sirius uses DASH APIs, whereas VFP uses layer-group-rule abstraction. Second, Sirius also benefits from a specialized ASIC implementation. Sirius is an NVA, which receives tunneled networking traffic from different endpoints aggregated by the physical network. In contrast, ROE evaluates these SDN policies on the node itself, which removes the need to have an additional hop on the physical network. Moreover, ROE supports the most common SDN policy abstraction used in Azure, providing a nearly effortless migration to existing VMs. Note that both approaches offer comparable latency on the VNET setup.

6.2 Resource Efficiency

ROE is designed to be area-efficient for FPGA implementations and employs various optimizations. To evaluate the impact of these optimizations, we synthesize the ROE hardware components on AMD Versal FPGAs. Table 3 compares the resource utilization against a range of academic and industrial RISC-V cores [17]. We observe that ROE's LUT utilization is comparable to VexRiscV and PicoRV32 cores while register and BRAM utilization are 1.94× and 4.75× higher respectively. This is because register file sizes and cache sizes scale linealy in the number of threads and cores. ROE's resource utilization is significantly lower than other RISC-V cores, Flute and SweRV. ROE sub-system resource utilization is primarily driven by ROE-Cores, followed by the Unified Flow Creation Engine that requires 5443 LUTs, 5096 registers, and 5 BRAMs. Other components use fewer than 1000 LUTs each. ***Overall, the ROE processing sub-system consumes only 5% of total FPGA resources, leaving sufficient room for other components, such as AccelNet.***

Table 3: Resource comparison of ROE-Core and general-purpose RISC-V cores used as embedded control processors

	VexRiscV [25]	PicoRV32 [32]	**ROE**	Flute [4]	SweRV [9]
LUTs	3.8K	4.0K	**4.8K**	14.2K	30.1K
Regs	3.3K	3.3K	**6.4K**	11.0K	16.2K
BRAMs	4	4	**19**	36	62

6.3 Impact of RISA

RISA program sizes depend on the SDN policy complexity, including number of layers, groups, rules, supported actions, stateful versus stateless behavior, encapsulation types, and search algorithms selected for conditional paths. Figure 7 shows that while static RISA program sizes scale rapidly with SDN policy complexity, dynamic instruction counts executed per packet grow at a much slower pace, confirming that higher policy expressiveness mainly impacts static instruction footprint rather than dynamic execution cost.

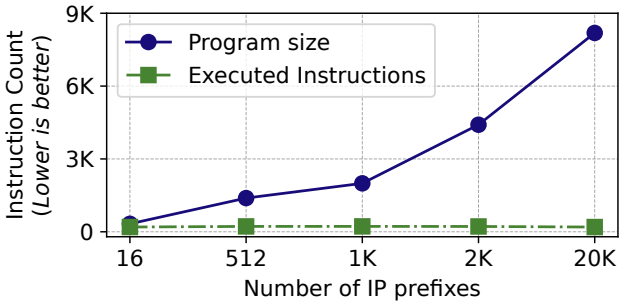


Figure 7: Increasing the number of IP prefixes increases the size of ROE programs (using a trie-based classifier) while number of executed instructions remains nearly constant.

Table 4 shows the instruction-type breakdown for a representative Azure VNET workload. The static RISA program contains ~92K instructions but only ~2.5K instructions are executed per packet on average. The dynamic instruction mix is dominated by `CTRL` and `MATCH` instructions, reflecting the prevalence of range-based and priority-ordered rule evaluation in SDN policies. Most `MATCH` and `ALU` operations use immediate operands, aligning well with our intent in RISA design. HASH operations are seldom used and dedicated logic in ROE-Core reduces their latency to only 1 cycle.

Table 4: Instruction distribution of a typical ROE program trace. As MATCH and CTRL operations dominate the trace, dedicated RISA instructions and an optimized ROE-Core implementation improve performance.

MATCH		ALU		CTRL	HASH	μENGINE
imm	reg	imm	reg			
31059	34	3091	545	48836	34	8278

RISA supports frequent SDN policy updates by recompiling programs while we continue executing previous versions. In our deployment, recompilation takes about 400 ms, with an additional 20 ms for mapping installation. Updates do not stall packet processing and CPS remains stable aside from brief cache warm-ups.

6.4 Impact of Caching Policies and Scaling

Connection establishment throughput scales in the number of threads and cores but is eventually limited by the memory bandwidth between L1 and L2 caches as well as their organization. To study this, we conduct experiments where we vary the L1 and L2 cache parameters. Our studies show that even after doubling the capacity of the L2 cache, its miss rate shows negligible improvements. In contrast, L2 miss rates reduce when the L2 cache is statically partitioned between VMs while keeping the capacity the same as the baseline, as shown in Figure 8. This highlights that thrashing limits overall L2 performance in the presence of multiple VMs.

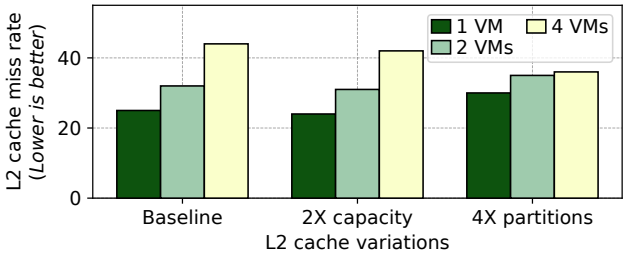


Figure 8: Impact of L2 cache organization on miss rate for different numbers of VMs. Increasing the cache capacity has negligible impact on miss rates. In contrast, while partitioning the cache increases miss rates in scenarios with fewer VMs, the miss rates decreases as the number of VMs increases.

We also conducted another experiment where we reduce the L1 line size by half (128B) and change the indexing policy of the L2 cache to use a hash function while other parameters, including L2 line size remain unchanged (256B). The L2 miss rate reduces substantially due to more efficient utilization of the available bandwidth and better handling of conflict misses, as shown in Figure 9.

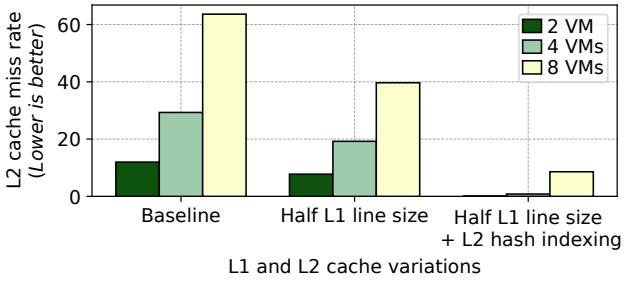


Figure 9: Impact of reducing L1 line size by half and changing the indexing function of L2 to use a hash function significantly reduces L2 miss rates.

7 Deployment

ROE is deployed at scale in production as a part of the Azure networking stack that supports SDN policy enforcement. ***At the time of this writing, ROE is enabled for approximately 18% of the tenant user base for the second generation of Azure Boost, serving production traffic across multiple regions and customer subscriptions.*** ROE is deployed on thousands of nodes. Across this fleet, ROE manages up to eight million active flows per node and sustains high rates of flow creation and deletion driven by VM lifecycle events, policy updates, and dynamic traffic patterns, while maintaining line-rate packet processing for steady-state traffic.

The system has been in operation for about a year and has progressed through different phases of offering, from an initial limited offering to a broader adoption. During this period, ROE has remained operational through multiple hardware and software revisions, incremental feature deployments, and compatibility upgrades, providing visibility into both steady-state performance and behavior under policy churn, partial upgrades, and failure recovery. ROE was introduced incrementally rather than as a clean-slate replacement and has coexisted with legacy software-based enforcement paths, mixed hardware generations, and heterogeneous tenant configurations. As a result, the deployment reflects realistic production conditions. The lessons described in this paper, particularly around hardware–software co-design, safe failure modes, and operational complexity, emerged directly from operating ROE at scale.

8 Lessons Learned

Reconfiguration capability of FPGAs accelerates time-to-market: Azure Boost initially contained only a minimum viable product mandatory to ensure functional completeness with pre-existing Azure SDN offerings. ROE hardware was added incrementally through subsequent firmware rollouts. In a typical silicon program, any feature request to ROE would have resulted in unforeseen program delays that would severely impact its tape-out and a cascading effect would have led to a longer time-to-market. However, the reconfigurability of FPGAs allowed us to provide AccelNet services on Azure in a short time, while ROE features were added and enhanced, often using lessons from prior deployments.

Deployment itself is a first-class feature in cloud: Deploying the correct components is necessary but insufficient; upgrade sequencing and cross-stack compatibility are equally critical. RoE interacts with VFP, Accelnet, Networking-Backplane components, and FPGA. Each component is deployed using their respective software delivery vehicle or agent in the cloud. Although ROE supports incremental production environments, its rollout exposed difficulty coordinating among several other agent updates. Production deployments showed that individually validated component versions could still fail in a mixed-version configuration that appeared only during real-world upgrade sequences at production scale. The root causes of these issues were often present in component interactions and compatibility dependencies rather than a single defective entity. These experiences reinforced the following best practices: (1) treat deployment ordering as a first-class design concern, (2) validate supported upgrade sequences and mixed-version states, and (3) include servicing, rollback, and recovery scenarios in scale testing.

Export health signals and telemetry to detect any regressions: Health signals and compatibility telemetry detected fleet-wide failures quickly. Staged rollouts, automated health gates, and the ability to pause or selectively disable functionality were essential to limit the blast radius of impact in a faulty build. These safeguards allowed ROE to expand safely despite compatibility issues that were difficult to reproduce before deployment.

Cloud workloads are often the best tests for the system: The cloud networking environment is complex and it runs several services in VMs or by the provider itself. Often, VM lifecycle events, such as VM shutdown or reboot, especially under high networking throughput exposed a few critical issues in the system, prompting us to rethink high-risk test scenarios. Focusing on these scenarios in early design cycles protected expensive and lengthy debugging time after deployment. Many critical issues appeared only when ROE was exercised through synthetic scenarios mimicking customer workloads involving policy updates, flow lifecycle transitions, VM mobility or servicing, and interactions with existing AccelNet components. These scenarios exposed implementation gaps when flow re-evaluation and mapping updates during high throughput traffic.

Operational simplicity and design abstraction matter as much as performance: Although ROE delivered substantial performance gains, operational complexity across hardware-software boundaries initially hindered adoption. API versioning, partial enablement, and policy churn created disproportionate operational overhead when interfaces were treated as static contracts. Designing these boundaries as evolving APIs - with clear invariants, compatibility rules, and rollback strategies-proved critical.

Safe degradation is required for serviceability: ROE may not function during software driver updates. For example, it can lose its ability to evaluate SDN policies during a RISA-Compiler upgrade. Therefore, hardware acceleration through ROE is not always available for every new connection. To handle this scenario, we ensure that cases which cannot be handled by ROE are processed through VFP-deferral, albeit with lower processing ability. This fallback allowed incremental rollout without forcing a disruptive reset of active traffic. The operational lesson is that production accelerators should be designed with explicit degradation modes from the beginning.

Verification plays a substantially different and critical role: FPGA and silicon development both rely on hardware description languages for RTL development and verification of these designs for quality assurance. For any conventional silicon product, verification accounts for a substantial fraction of the overall development time to avoid expensive re-spins. In contrast, FPGA firmware can be rebuilt within hours, enabling software integration testing as soon as new builds are available. We leverage this property by incorporating software-driven validation test cases after the verification environment has achieved moderate coverage targets. This enables faster RTL-to-deployment iteration than the RTL-verification loop typical of silicon designs, resulting in a faster time-to-market. This approach also provides significant run-time test coverage when scaled across multiple nodes, compared to slow simulation environments. Our experience suggests three key observations. *First*,

verification remains the primary mechanism for ensuring correctness prior to software integration. *Second*, owing to the complexity of system debug in Azure, verification remains paramount for reproducing any failure observed during deployment or validation to simplify debugging. *Third*, while validation frameworks cover common execution paths, many error conditions and corner cases are tedious to generate and track; these are better addressed in the verification environment. Consequently, FPGA's software-like development methodology shortens software integration cycles, without compromising overall quality. At the same time, verification remains indispensable to program success, complementing validation by enabling seamless software integration and covering cases impractical to exercise through validation.

9 Related Work

ROE is designed, implemented, and deployed on Microsoft's FPGA-based SmartNIC devices on top of its AccelNet platform [14]. While AccelNet primarily focuses on fast processing of packets corresponding to already established connections, ROE takes one step further and offloads the new connection establishment processing onto SmartNIC devices. ROE is a direct extension of Microsoft's VFP [13] software. ROE accelerates VFP for common scenarios with improved performance and full feature parity, including not only rule processing but also monitoring and servicing.

Prior work from industry on first-time packet processing typically involves dedicated gateway devices. Google's Andromeda [10] leverages a Hoverboard gateway device specialized for first-time packet processing. In fact, Microsoft also leverages a similar model using its Sirius gateway device [3] for certain network scenarios. In contrast, ROE is local to a data center server and can handle connection establishment before the packet actually leaves the server, reducing the cost of dedicated hardware. Industrial chip designers have introduced SmartNIC/DPU cards, such as Nvidia BlueField DPU [7], Intel IPU [15], and AMD FPGA-based SmartNIC [11]. Most SmartNIC devices target the processing of connections that have already been set up while leaving connection establishment processing to host or embedded processors, such as general-purpose ARM cores [19, 28]. In contrast, our work presents a highly specialized soft core on an FPGA that supports connection establishment for a wide range of network virtual functions.

Several priori academic works exist on programmable SmartNICs and switching architectures. Early frameworks like RMT [6] introduced flexible match-action pipelines for high-speed forwarding, while FlexNIC [18] showed NIC-level programmability that offloaded custom packet handling to reduce host overhead. These works primarily focus on accelerating packets for established connections and lack the ability to support complex, unbounded SDN policy expression. Later architectures, such as dRMT [8] and FlexCore [30], aimed to replace fixed pipelines with run-to-completion processors and dynamic runtime reconfiguration, enabling them to support first-time packet processing. In contrast, ROE introduces a dedicated processing complex that operates in parallel with a bounded-latency match-action pipeline (AccelNet). This separation enables independent optimizations for both established and new connections using fundamentally different execution models, without interference between them.

The design of RISA builds upon prior languages for programmable NICs. It adopts the match-action abstraction popularized by P4 [5], but expresses these semantics at the instruction level, leveraging a register file for parsing and template-based classifiers for lookup. Unlike P4 and other pure data-plane languages, such as Domino [23], RISA does not impose bounded execution constraints, enabling more flexible control flow. While other instruction sets, such as NetASM [22], were proposed as an intermediate representation to bridge high-level languages and heterogeneous hardware targets, RISA differs fundamentally in its design goals. Specifically, RISA eliminates load/store instructions by embedding lookup tables directly into the instruction stream and introduces CISC-style match instructions to efficiently express complex logical conditions. These design choices make RISA a specialized, performance-oriented language for first-packet SDN policy evaluation, rather than a general-purpose, portable data-plane language.

10 Conclusion

Software Defined Networking (SDN) is critical in public cloud environments. For SDN policy evaluation, Microsoft Azure primarily relies on two specialized platforms: software-based Virtual Filtering Platform (VFP) for connection setups and an FPGA-based accelerator, called AccelNet, to quickly process subsequent packets. However, slow VFP processing limits connection establishment rates and acts as a critical barrier to scaling the performance of Azure cloud networking infrastructure. This paper presents *Rules Offload Engine (ROE)* that addresses this challenge.

ROE is a hardware-software co-designed accelerator implemented on FPGA-based SmartNICs. It features a networking optimized ROE Instruction Set Architecture (RISA), a software control plane that orchestrates communication with existing AccelNet and VFP, a specialized just-in-time compiler that translates SDN policies into RISA programs, a core featuring several micro-architectural optimizations for efficient FPGA implementation, and the necessary modules to seamlessly integrate into existing Azure ecosystem. Our evaluations show that ROE achieves up to 400K CPS connection establishment throughput, yielding more than 10x improvement over prior methods. ROE is deployed in Azure (enabled on $\sim$ 18% of the tenant user base) and was featured as a part of the second generation of Azure Boost.

Acknowledgments

We dedicate this paper to the memory of our beloved colleague, friend, and mentor, Derek Chiou. As a founding member of the Accelnet program and a strong advocate for reconfigurable accelerators, Derek inspired, supported, and encouraged us to push the boundaries of hardware-software co-design. We thank our current and past collaborators—Manasi Deval, Hrishi Tidke, Georgia Walker, Taylor Swanson, Dennis Gurzhii, Vincent Nguyen, Aniket Dhobe, Kumara Rathinavel, Khoa To, Roi Aibester, Guy Gini, and Priyanka Gandhara—for their support and contributions to the ROE program. We also thank Boris Bobrov, Max Uritsky, and Arun Kishan, for believing in us and providing us with the resources to execute this program. We thank our reviewers and shepherd for their valuable feedback, which helped us improve the paper.

This work does not raise any ethical issues.

References

[1] Ahmed Abdelsalam, Osman Nuri Ertugay, Zachary Noel Libby, Narayanan Ravichandran, Rohan Kandi, Milan Dasgupta, Anshuman Verma, Lok Chand Koppaka, and Harish Srinivasan. 2026. Forceful flow termination for accelerated networking. (March 5 2026). US Patent App. 18/821,226.

[2] Amazon [n. d.]. *Amazon Web Services*. Amazon. https://aws.amazon.com/

[3] Deepak Bansal, Gerald DeGrace, Rishabh Tewari, Michal Zygmunt, James Grantham, Silvano Gai, Mario Baldi, Krishna Doddapaneni, Arun Selvarajan, Arunkumar Arumugam, Balakrishnan Raman, Avijit Gupta, Sachin Jain, Deven Jagasia, Evan Langlais, Pranjal Srivastava, Rishiraj Hazarika, Neeraj Motwani, Soumya Tiwari, Stewart Grant, Ranveer Chandra, and Srikanth Kandula. 2023. Disaggregating Stateful Network Functions. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1469–1487. https://www.usenix.org/conference/nsdi23/presentation/bansal

[4] Inc Bluespec. 2018. Flute, a free and open-source RISC-V CPU. https://github.com/bluespec/Flute. (2018).

[5] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95. https://doi.org/10.1145/2656877.2656890

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

[7] Idan Burstein. 2021. Nvidia data center processing unit (dpu) architecture. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–20.

[8] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/3098822.3098823

[9] Western Digital Corporation. 2019. VeeR EH1 RISC-V Core. https://github.com/chipsalliance/Cores-VeeR-EH1. (2019).

[10] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[11] Jaideep Dastidar. 2023. Fpgas and their evolving role in domain specific architectures: A case study of the AMD 400G adaptive smartNIC/DPU soc. In *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. 151–151.

[12] Osman Ertugay. 2025. NCPS - Network Connection Performance Stress Tool. https://github.com/microsoft/ncps. (2025).

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

[14] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[15] Pat Fleming, Chihjen Chang, Derek Collier, Anjali Singhai, Stephen Doyle, Eliel Louzoun, David Lee, Vetrivel Ayyavu, Sarig Livne, Robert Hathaway, Tony Hurson, Jackson Ellis, Tamar Bar-Kanarik, Jonathan Kenny, Cristine Dumitrescu, and Yaron Wolberger. 2025. Intel® IPU E2200: Second Generation Infrastructure Processing Unit (IPU). In *2025 IEEE Hot Chips 37 Symposium (HCS)*. 1–16. https://doi.org/10.1109/HCS66204.2025.11154402

[16] Google [n. d.]. *Google Cloud Platform*. Google. https://cloud.google.com/

[17] Carsten Heinz, Yannick Lavan, Jaco Hofmann, and Andreas Koch. 2019. A catalog and in-hardware evaluation of open-source drop-in compatible RISC-V softcore processors. In *2019 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. IEEE, 1–8.

[18] Antoine Kaufmann, Simon Peter, Naveen Kr Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High performance packet processing with flexnic. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. 67–81.

[19] Elie F Kfoury, Samia Choueiri, Ali Mazloum, Ali AlSabeh, Jose Gomez, and Jorge Crichigno. 2024. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access* (2024).

[20] Microsoft [n. d.]. *Microsoft Azure*. Microsoft. https://azure.microsoft.com/

[21] NVIDIA [n. d.]. *Mellanox: ConnectX-7 400G Adapters Accelerated networking for modern data center infrastructures*. NVIDIA. https://nvdam.widen.net/s/srdqzxgdr5/connectx-7-datasheet

[22] Muhammad Shahbaz and Nick Feamster. 2015. The case for an intermediate representation for programmable data planes. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. 1–6.

[23] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet transactions: High-level programming for line-rate switches. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 15–28.

[24] Burton J Smith. 1982. Architecture and applications of the HEP multiprocessor computer system. In *Real-Time signal processing IV*, Vol. 298. SPIE, 241–248.

[25] SpinalHDL. 2017. VexRiscv. https://github.com/SpinalHDL/VexRiscv. (2017).

[26] Harish Srinivasan, Tan Tian, Milan Dasgupta, and Denys Gurzhii. 2025. Data packet traffic using speculative unified flow. (Nov. 11 2025). US Patent 12,470,483.

[27] Tan Tian, Anshuman Verma, Garg Tushar, and Taylor Catherine Swanson. 2024. System and method for processing networking packets using a reorder queue. (Dec. 19 2024). US Patent App. 18/337,403.

[28] Nathan Tibbetts, Sifat Ibtisum, and Satish Puri. 2025. A Survey on Heterogeneous Computing Using SmartNICs and Emerging Data Processing Units (Expanded Preprint). *arXiv preprint arXiv:2504.03653* (2025).

[29] Zeke Wang, Hongjing Huang, Jie Zhang, Fei Wu, and Gustavo Alonso. 2022. FpgaNIC: An FPGA-based Versatile 100Gb SmartNIC for GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 967–986. https://www.usenix.org/conference/atc22/presentation/wang-zeke

[30] Jiarong Xing, Kuo-Feng Hsu, Matty Kadosh, Alan Lo, Yonatan Piasetzky, Arvind Krishnamurthy, and Ang Chen. 2022. Runtime programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. 651–665.

[31] Jiarong Xing, Yiming Qiu, Kuo-Feng Hsu, Songyuan Sui, Khalid Manaa, Omer Shabtai, Yonatan Piasetzky, Matty Kadosh, Arvind Krishnamurthy, T. S. Eugene Ng, and Ang Chen. 2023. Unleashing SmartNIC Packet Processing Performance in P4. In *Proceedings of the ACM SIGCOMM 2023 Conference (ACM SIGCOMM '23)*. Association for Computing Machinery, New York, NY, USA, 1028–1042. https://doi.org/10.1145/3603269.3604882

[32] YosysHQ. 2015. PicoRV32- a size-optimized RISC-V CPU. https://github.com/YosysHQ/picorv32. (2015).

A SDN Policy Example

Algorithm 1: Example: SDN policy translation to classifiers

if	
srcIP ∈ 10.1.0.0/16	LPM
and srcPort ∈ [10000, 20000]	Interval Tree
and dstIP == 192.168.1.10	Hash Lookup
and dstPort == 443	Hash Lookup
and protocol ∈ {TCP, UDP}	Linear Search
then	
allow	
else	
block	
An incoming packet:	
srcIP : 10.1.2.3	LPM /16
srcPort : 15000	Found in range
dstIP : 192.168.1.10	Bucket hit
dstPort : 443	Bucket hit
protocol : TCP	TCP match
action executed:	
allow	

B RISA Translation of the SDN Policy Example

```
# RISA Assembly Program: SDN Policy Translation to Classifiers
# Register Usage:
# REG_0 (RISA_REGISTER_ZERO) - ZERO
# REG_2 (RISA_REGISTER_HCR) - Header control
# REG_8 : REG_31 (RISA_REGISTER_GPRs) - General purpose
# REG_40 (RISA_REGISTER_SOURCE_IPV4_OR_SOURCE_IPV6_1) - Source IP
# REG_44 (RISA_REGISTER_DEST_IPV4_OR_DEST_IPV6_1) - Destination IP
# REG_51 (RISA_REGISTER_SOURCE_PORT) - Source Port
# REG_52 (RISA_REGISTER_DEST_PORT) - Destination Port
# REG_48 (RISA_REGISTER_IP_PROTOCOL) - IP Protocol

# Constants:
# 10.1.0.0 = 0x0A010000; /16 mask = 0xFFFF0000
# 192.168.1.10 = 0xC0A8010A; TCP = 6; UDP = 17

# Set HCR to ONE to point to innermost original header (Header 1)
ALU_IMM_MOV REG_2 1

# CHECK 1: srcIP \in 10.1.0.0/16 (LPM - Longest Prefix Match)
# Extract upper 16 bits of srcIP and compare with 10.1 (0x0A01)
# srcIP[31:16] should equal 0x0A01 for a match
# Load source IP into working register (MOV is OR with REG_0)
ALU_REG_OR REG_9 REG_40 REG_0

# Shift right by 16 to get upper 16 bits (10.x.x.x -> 0x0A01xxxx >> 16 = 0x0A01)
ALU_IMM_SHR REG_9 REG_9 16

# Compare with 0x0A01 (10.1 network prefix)
# WRWR: Write to SCR and MCR
# If srcIP prefix == 0x0A01, MCR becomes 1 (match)
# Syntax: MATCH_IMM_TYPE RS ImmediateValue ShiftAmount
MATCH_IMM_WRWR_EQ_SHIFT REG_9 0x0A01 0

# CHECK 2: srcPort \in [10000, 20000] (Interval Tree - Range Check)
# Need: srcPort >= 10000 (0x2710) AND srcPort <= 20000 (0x4E20)

# Load source port (MOV is OR with REG_0)
ALU_REG_OR REG_10 REG_51 REG_0

# First check: srcPort >= 10000 (0x2710)
# ANDAND: AND result with current MCR
MATCH_IMM_ANDAND_GTEQ_SHIFT REG_10 0x2710 0

# Second check: srcPort <= 20000 (0x4E20)
# ANDAND: AND current SCR with MCR, AND result with MCR
MATCH_IMM_ANDAND_LTEQ_SHIFT REG_10 0x4E20 0

# CHECK 3: dstIP == 192.168.1.10 (Hash Lookup - Exact Match)
# 192.168.1.10 = 0xC0A8010A
# Load destination IP (MOV is OR with REG_0)
ALU_REG_OR REG_11 REG_44 REG_0

# Compare with 192.168.1.10 (0xC0A8010A)
# ANDAND: AND current SCR with MCR, AND result with MCR
MATCH_IMM_ANDAND_EQ_SHIFT REG_11 0xC0A8010A 0

# CHECK 4: dstPort == 443 (Hash Lookup - Exact Match); # 443 = 0x01BB
# Load destination port (MOV is OR with REG_0)
ALU_REG_OR REG_12 REG_52 REG_0

# Compare with 443 (0x01BB)
# ANDAND: AND current SCR with MCR, AND result with MCR
MATCH_IMM_ANDAND_EQ_SHIFT REG_12 0x01BB 0

# CHECK 5: protocol \in {TCP(0x6, 6), UDP(0x11, 17)} (Linear Search)
# Check if protocol == TCP OR protocol == UDP
# Load protocol (MOV is OR with REG_0)
ALU_REG_OR REG_13 REG_48 REG_0
```

C RISA Policy Classifier

```


# First check: protocol == TCP (6)
# WRNOP: Write to SCR only (don't modify MCR yet)
MATCH_IMM_WRNOP_EQ_SHIFT REG_13 0x6 0


# Second check: protocol == UDP (17)
# ORWR: OR result into SCR, write to MCR
# This creates: SCR = (protocol==TCP) OR (protocol==UDP)
MATCH_IMM_ORWR_EQ_SHIFT REG_13 0x11 0


# Now AND the protocol match result with the accumulated MCR
# Use ANDAND to combine: MCR = MCR AND SCR
# Check if SCR is non-zero (NEQ 0)
MATCH_IMM_ANDAND_NEQ_SHIFT REG_0 0x0 0

############################################################
# DECISION: Based on MCR value, ALLOW or DROP
#
# If MCR == 1: All conditions matched -> Allow packet (jump to ALLOW)
# If MCR == 0: At least one condition failed -> Fall through to DROP
############################################################

# Jump to ALLOW if MCR is set (all conditions passed)
CTRL_JMPR_IFMCR REG_0 _ALLOW

# DROP: Block the packet (fall through case when MCR == 0)
ENGINE_OUTCOME_DROP_TERM

# ALLOW: Create a new flow
_ALLOW:
ENGINE_UF_CREATE
# Forward the packet (terminate successfully)
CTRL_TERM

############################################################
# End of RISA SDN Policy Classifier Program
############################################################
```