USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

HybridMesh: A Hardware-software Hybrid Approach for Accelerating Service Mesh Ingress

Myoungsung You, *University of Seoul;* Jaehyun Nam, *Dankook University;*
Minjae Seo, *ETRI;* Taejune Park, *Chonnam National University;* Seungwon Shin, *KAIST*

https://www.usenix.org/conference/nsdi26/presentation/you

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

HybridMesh: A Hardware-software Hybrid Approach for Accelerating Service Mesh Ingress

Myoungsung You
University of Seoul

Jaehyun Nam
Dankook University

Minjae Seo
ETRI

Taejune Park
Chonnam National University

Seungwon Shin
KAIST

Abstract

Service meshes have become essential for enabling microservice communication in cloud environments. Despite their benefits, they impose significant network overhead. We identify that the ingress gateway, the main entry point for external traffic, has emerged as a major performance bottleneck. This overhead stems from two key limitations of current ingress gateways: (1) CPU-intensive traffic analysis and (2) a complex packet forwarding path. Our analysis shows that these inefficiencies can reduce network throughput by up to 4× while substantially increasing CPU utilization. In response, we propose *HybridMesh*, a hardware-software hybrid ingress gateway that leverages a SmartNIC for high-performance traffic analysis and efficient traffic routing. This process is augmented by a lightweight CPU-side proxy that provides various traffic management features not suitable for SmartNIC execution. Evaluations show that *HybridMesh* outperforms existing ingress gateways, achieving a 4.4× increase in HTTP throughput. Furthermore, compared to software-only and hardware-only designs, our hybrid design delivers 2.3× higher cost-effectiveness and 2.8× better power-efficiency.

1 Introduction

Service meshes [51] are becoming the de facto standard communication infrastructure for implementing large-scale microservices on cloud environments. A recent survey revealed that 47% of organizations deploy their applications on service meshes [2]. Operating as an intermediate layer, service meshes decouple communication logic between microservices, such as peer discovery or load-balancing, from applications. This separation simplifies the application development process and enhances the manageability of microservices.

Service meshes, while central to facilitating microservice communication, often introduce substantial network overhead due to their inherent complex design. A notable example of such overhead is attributed to sidecars [34]. Sidecars are reverse proxies deployed alongside each application to manage inter-service communication. These sidecars create extra network hops that can degrade network performance. While most previous studies on service meshes [62, 63, 72] have focused on reducing sidecar overhead through methods such as shared-memory data channels or kernel offloading, our research identifies a more crucial bottleneck: *the ingress gateway*. In service mesh environments, the ingress gateway acts as the main entry point for external traffic, routing incoming packets to the appropriate destination sidecars and managing essential traffic management features such as circuit breaking, load balancing, and failovers. Despite its pivotal role, current ingress gateways struggle to achieve high performance and scalability due to the following two main challenges.

Challenges. The first one is the *CPU-intensive traffic analysis task*. Since most microservices employ application-layer (L7) protocols such as HTTP REST APIs [53], accurately dispatching external packets requires deep parsing of variable-length L7 headers and matching the parsed fields against numerous ingress policies. This process often involves multiple regular expression (regex) matches and imposes a considerable burden on ingress gateways, which must process all external packets entering the service mesh. The second challenge is the *complex packet forwarding path*. In cloud environments, ingress gateways and applications usually run within isolated execution units (e.g., containers). This isolation forces traffic exchanges to traverse intricate forwarding paths involving virtualization layers, resulting in repeated packet copies and redundant network stack operations. Our evaluation shows that these two factors severely degrade ingress gateway performance, reducing throughput by 4× even under only 100 concurrent clients with a single routing policy (§ 2.2).

Our approach. To overcome the above challenges, we present *HybridMesh*, a novel hardware-software hybrid ingress gateway designed for seamless integration within the current service mesh architecture. *HybridMesh* utilizes the capabilities of a SmartNIC [52] to accelerate CPU-intensive traffic analysis essential for dispatching external packets to their destination services. Based on the processing results of the SmartNIC, a lightweight CPU-side proxy provides the

remaining traffic management features, which are beyond the capabilities of SmartNICs alone. *HybridMesh* also employs a *HybridMesh* tunnel that optimizes the complex forwarding path between an ingress gateway and destination services through eBPF-based interface merging and redirection. This hybrid design offers the same functionalities as existing software gateways while benefiting from hardware acceleration.

In our system, external packets are first processed by the *HybridMesh NIC stack*, implemented on the SmartNIC. It employs parallel traffic analyzers that inspect packet headers and payloads to identify the appropriate service-level destinations. A specialized allocation algorithm further optimizes this step by distributing packets across analyzers according to their connection state. The NIC stack then encodes the analysis results into each packet as metadata for later stages. After this NIC stack processing, the *HybridMesh node stack* takes over, using the per-packet metadata together with the network state of co-located containers to perform container-level routing. At this stage, it also enforces traffic management policies, enabling various deployment scenarios such as canary releases and circuit breaking, without repeating CPU-intensive traffic analysis. Packet exchanges, which occur from the NIC stack to the node stack and subsequently to containers, are coordinated by the *HybridMesh tunnel*. The tunnel supports SR-IOV-based [1] direct transfers between NIC and node stacks as well as socket-level redirection between the node stack and containers. This design avoids redundant user-kernel crossings and thereby shortens the forwarding path.

Contributions. We make the following contributions:

- We analyze the limitations of the current service mesh data plane, focusing on traffic ingress processing.
- We design and implement *HybridMesh*, a novel hardware-software hybrid ingress gateway that offloads traffic analysis to a SmartNIC while preserving rich traffic management features via a lightweight CPU-side proxy.
- We evaluate *HybridMesh* on a commodity SmartNIC [3] and show that it achieves up to 4.4× higher throughput and 2× lower latency than existing ingress gateways, as well as 2.3× greater cost-effectiveness and 2.8× higher power-efficiency than software-only and hardware-only designs.

2 Background and Motivation

2.1 Service Mesh Architecture

Service meshes [51] decouple complex inter-service communication logic from the application side, providing an intermediate layer for microservices. Figure 1 shows a common service mesh architecture. A service mesh consists of a set of sidecar proxies. A sidecar is deployed alongside an application container, and they are packaged into a single pod [13], a management unit of container environments. The sidecar and

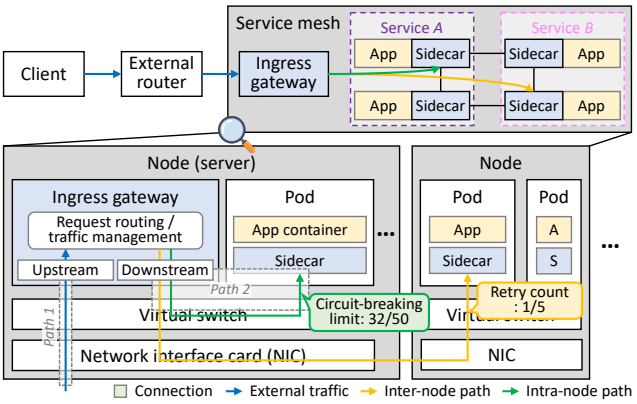

Figure 1: A common service mesh architecture.

the app container then share the same resources inside of the namespace of the pod [29]. Whenever a container sends or receives packets, a sidecar intercepts the packets and provides various functionalities required for service communication (e.g., service discovery) on behalf of the container.

In service meshes, pods are grouped into a *service* abstraction [4] as shown in Figure 1. Each service represents a set of pods running the same application, allowing for efficient load distribution. To enable external access to these internal services, service meshes employ an ingress gateway. As shown in Figure 1, this gateway is deployed on a node within the service mesh acting as the primary entry point for external traffic from clients. Specifically, it routes incoming request packets to their respective destination services, relaying packets from downstream connections (between clients and the gateway) to upstream connections (between the gateway and pods).

Upon receiving request packets, the ingress gateway conducts a detailed analysis to identify services matched to the packets. Given the prevalent utilization of L7 protocols (e.g., HTTP) by microservices [37,44,59], the gateway inspects the headers and payloads of the incoming packets to determine the correct services. This process involves parsing various HTTP fields and a number of regex matches against the parsed fields based on the ingress policies that contain matching criteria (e.g., URLs). Once the service is identified, the ingress gateway selects an appropriate endpoint pod from the available pods within the service. This pod selection takes into account the state of individual pods (e.g., connection loads) or predefined conditions (e.g., HTTP headers), enabling various load-balancing strategies, such as canary deployment.

Once an endpoint pod is selected, the ingress gateway transmits packets via an upstream connection linked to the pod (depicted as the green arrow in Figure 1). If the endpoint pod is located on a different node from the ingress gateway (the orange arrow), the packet is encapsulated and forwarded to the destination node through the network interface card (NIC). During this upstream processing, the ingress gateway performs various *traffic management features* required for communication with the pod. As shown in Figure 1, it lim-

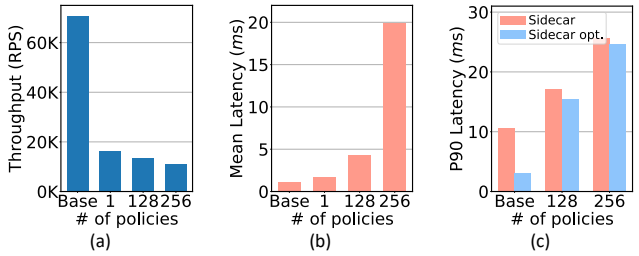

Figure 2: Performance of Nginx pods under varying numbers of ingress policies. *Base* means no ingress gateway actions.

its the maximum concurrent requests to the pod to prevent overloading (i.e., circuit breaking [41]). In addition, if the pod is unable to accept a new request due to its current load, the gateway holds the request and retries it within a predefined threshold (i.e., retries [43]). By performing these traffic management features, the ingress gateway ensures that external requests are reliably dispatched to their destinations, even in the presence of temporary pod or network failures.

2.2 Challenges in Service Mesh

While a service mesh decouples service communication from application logic, its complex architecture introduces non-trivial performance overheads [72]. Most prior work focuses on mitigating sidecar-induced overhead, which adds extra network processing within and across pods. For example, Ambient mesh [10] optimizes communication between app containers and sidecars by relocating sidecar functionality into kernel space, thereby reducing redundant user-space processing and user–kernel transitions. However, optimizing sidecars alone does not eliminate all performance bottlenecks in the data plane. We observe that the ingress gateway can become an equally critical bottleneck. Unlike sidecars, which enforce policies in a distributed, per-pod manner, the ingress gateway processes all external traffic at a centralized point while maintaining a broader set of routing and traffic management policies. This centralized design makes the gateway particularly susceptible to congestion under high load. Moreover, delays introduced at the gateway propagate to downstream components, potentially causing head-of-line blocking that affects end-to-end latency. Nevertheless, existing ingress gateways [8, 11, 14] often struggle to sustain high performance.

To evaluate this limitation, we conduct experiments on an Istio service mesh [12]. Eight Nginx pods are deployed as a single Kubernetes service, and an Istio ingress gateway [11] distributes HTTP traffic based on URL patterns. To isolate gateway overhead from sidecar overheads, we replace user-space sidecars with lightweight in-kernel sidecars commonly used in sidecar optimization studies [6, 10, 62]. Details of the evaluation setup are provided in § 5.1. Figure 2 shows the results under 100 concurrent connections. When requests are sent directly to pods, throughput (RPS) is significantly higher than when routed through the ingress gateway. As shown in Figure 2(a), even a single URL-matching policy reduces throughput by 4.3×. Under a 5K RPS workload, mean latency increases by 1.5×, as shown in Figure 2 (b). Scalability further degrades with more policies: with 256 URL rules, throughput decreases by 6.5× and latency increases by 18×.

Table 1: CPU usage and details of each task within an ingress gateway workflow.

Task	*Detail*	*CPU*	*Type*
Downstream processing	Establishing and maintaining TCP/IP sessions with users	5.64%	Control-intensive
Traffic analysis	**Packet parsing, L3/L4 matching, regex-based L7 matching**	**39.22%**	**Compute-intensive**
Traffic management	Routing and resilience policy enforcement and error handling	8.78%	Control-intensive
Upstream processing	Establishing and maintaining TCP/IP sessions with pods	5.96%	Control-intensive
Others	Logging and runtime management	30.4%	-

Moreover, the ingress gateway can diminish the benefits of sidecar optimization. Figure 2 (c) compares tail latency at 1K RPS with and without sidecar optimization. Without the gateway, optimizing sidecars reduces tail latency by avoiding redundant kernel and user space processing. Once the gateway is introduced, however, tail latency rises to 25.66 ms, reverting to a level comparable to the non-optimized configuration. This occurs because congestion at the centralized gateway induces head-of-line blocking that dominates end-to-end latency. The effect is further amplified by resource contention: processing only 128 rules causes the ingress gateway to consume more than 40% of CPU resources, reducing the capacity available to other components. These results demonstrate that the ingress gateway becomes a severe bottleneck at the entry point of external request packets and even diminishes the benefits of sidecar optimization. We further analyze this problem and identify the following two root causes.

CPU-intensive traffic analysis. The first issue with the current ingress gateway design is its CPU-intensive nature. Microservices commonly utilize L7 protocols, such as REST APIs, which are crucial for effective inter-service communication [37]. The process of routing L7 packets to their designated pods requires comprehensive parsing and analysis of packet headers and payloads by the ingress gateway. Specifically, routing HTTP traffic involves inspecting various parameters, including port numbers, host, paths, and user-defined fields. Also, the operational burden on the ingress gateway is further intensified by the need to manage a large number of policies, each corresponding to active services within the service mesh. This results in significant CPU utilization due to multiple regex matches applied across these policies.

To show the extent of this problem, we analyze the CPU usage of the Istio ingress [12]. Table 1 provides a detailed breakdown of CPU cycles consumed by the Istio ingress, as measured by *perf* [17]. Notably, the traffic analysis tasks dominate CPU usage, accounting for 39% of the total cycles. This

high consumption is attributed to the need for payload parsing across various HTTP fields and the extensive regex matching against these parsed fields. In comparison, tasks associated with providing traffic management features, such as circuit breaking and failover, involve direct network interaction with endpoint pods and account for less than 20% of CPU cycles. This example shows the predominant consumption of CPU cycles by L7 traffic analysis tasks. Offloading these CPU-intensive operations from the CPU side holds promise for enhancing the performance of ingress gateways.

Complex forwarding path. Ingress gateways also suffer from performance degradation due to their complex traffic forwarding path. In typical service mesh deployments, the gateway runs in a dedicated pod and connects to a virtual switch in the root namespace. While this design simplifies management and enables communication with other pods attached to the switch, it complicates packet forwarding. As shown in Figure 1, this path comprises two sub-paths: (1) from the NIC to the gateway and (2) from the gateway to endpoint pods. In the first sub-path, an incoming packet must traverse the virtual switch and undergo two network stack operations: one at the root namespace and another at the gateway's namespace. In the second segment, the packet forwarded by the gateway to the selected endpoint pod again passes through the virtual switch and two additional network stack operations on each pod's namespace. This process is mirrored in reverse when responses return from the pod to the gateway. Although some eBPF-based optimizations for sidecars [62] could be applied to optimize the second segment, no prior work has tackled the first segment. Consequently, the complex forwarding path induces repeated packet copies and context switches, thereby reducing throughput and increasing latency [36, 51].

2.3 Opportunities with SmartNICs

The challenges identified in the previous section underscore the ingress gateway as a performance bottleneck within a service mesh. While one might consider scaling out the ingress gateway by distributing the load across multiple instances on different nodes [20], this method falls short of addressing the core problem. Even under low overhead conditions with only 100 connections and a single policy, our evaluations (Figure 2) reveal that the current ingress gateway design degrades throughput by 4x, showing that merely increasing the number of gateway instances does not fundamentally resolve the performance limitations. As the number of connections and policies increases, the performance degradation worsens.

To tackle this problem, our approach shifts towards leveraging programmable devices for offloading key tasks of the ingress gateway. Among potential candidates, we opt for SmartNICs [30] due to their robust capabilities and popularity. Specifically, the Bluefield-2 SmartNIC [3], used in this study, features various accelerators for high-performance packet processing and 8 ARM cores with 16 GB of memory for executing control logic. Furthermore, SmartNICs are widely adopted in cloud environments [35, 57, 65], making them an ideal choice for enhancing ingress gateway performance.

Table 2: Performance of HAProxy with host and SmartNIC settings under 100 connections and a single routing rule.

Setting	*Throughput*	*Avg latency*	*P90 latency*
Host	10,502 RPS	9.52 *ms*	11.30 *ms*
SmartNIC	7,203 RPS	14.43 *ms*	21.52 *ms*

While previous studies have offloaded various L3/L4 network functions to SmartNICs to improve performance [32, 33, 54, 60, 70, 71], extending this approach to execute a complete ingress gateway on a SmartNIC presents additional challenges. As shown in Table 1, an ingress gateway includes both compute-intensive and control-intensive tasks. The latter requires complex connection state management and sophisticated error handling (e.g., TCP/IP processing and request buffering), making it better suited for host CPUs than for low-frequency SmartNIC cores. To show this limitation, we deploy HAProxy [8], a common building block for implementing an ingress gateway, on both the host OS and the SmartNIC. We then configure the HAProxy to route external traffic to the host's Nginx pod and measure its performance. As shown in Table 2, executing the entire HAProxy on the SmartNIC results in about a 30% decrease in performance compared to the case where it runs on the host OS due to the limited capability of the SmartNIC's cores. Thus, offloading the entire ingress gateway onto a SmartNIC is inefficient and fails to address the core performance bottleneck.

3 Design

As discussed in § 2.2 and § 2.3, ingress gateways involve both CPU-intensive tasks and complex tasks unsuitable for on-NIC processing. Thus, relying on a single processor to handle all ingress gateway tasks is inefficient. To address this challenge, *HybridMesh* uses a hardware-software hybrid design that partitions the ingress workflow across the CPU and the SmartNIC based on their characteristics. Specifically, *HybridMesh* offloads compute-intensive traffic analysis tasks to the SmartNIC, while retaining down/upstream processing and traffic management features on the CPU-side, which require control-intensive processing. This design segregates the ingress gateway's responsibilities into two key components: the *HybridMesh* (HM) NIC stack, which accelerates traffic analysis, and the HM node stack, which uses per-packet metadata from the NIC stack to route traffic and enforce traffic management policies without CPU-intensive tasks.

3.1 Overall Workflow

In our system, an incoming packet is first processed by the HM NIC stack located in the SmartNIC. ① As shown in Figure 3,

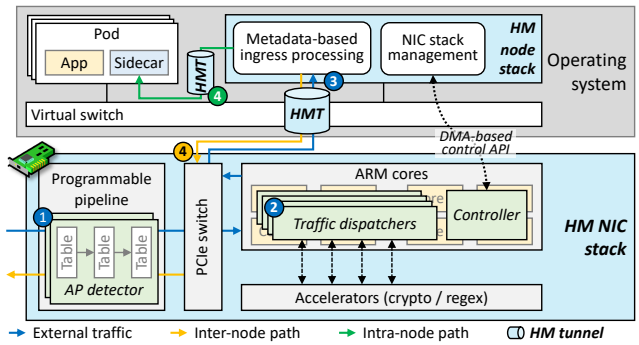

Figure 3: The overview of the *HybridMesh* system.

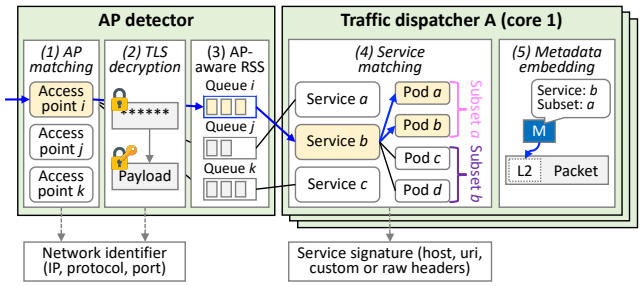

Figure 4: The HM NIC stack's workflow.

the access point (AP) detector, which is implemented on the SmartNIC's switching ASIC, performs network-level matching at line-rate, identifying service request packets among incoming packets. ② These request packets are then allocated to traffic dispatchers running on the on-NIC cores (e.g., ARM cores) based on their connection states. In TLS scenarios, encrypted packets are first processed by the kTLS offloading mechanism [58] to perform TLS record decryption before being assigned to dispatchers. The dispatchers operate in parallel, extracting various L7 fields from packets according to offloaded ingress policies to determine their destination pods. Hardware accelerators handle the actual policy matching tasks, ensuring high-performance processing. The dispatchers then encode the matching results as metadata within each packet before forwarding them to the CPU-side.

Packet forwarding paths on the CPU-side are optimized by the *HybridMesh* tunnel (HMT) that enables socket-level packet exchanges between *HybridMesh* components and pods. ③ Packets from the NIC stack are ingressed directly into the node stack's namespace via the HMT, bypassing the virtual switch. Upon reception, the node stack interprets the embedded metadata to finalize traffic routing and to provide traffic management features required for the endpoint pods, such as circuit breaking and failover strategies. This ensures that packets are precisely and efficiently routed to the designated endpoint pods, whether within the same node or across different nodes. ④ Intra-node traffic transfers directly, ④ while inter-node traffic reroutes through the NIC stack, both facilitated by the HMT to avoid redundant kernel processing. In the following sections, we explain each component of *HybridMesh* in the packet processing order.

3.2 *HybridMesh* NIC Stack

As shown in Figure 4, incoming packets are initially processed by the NIC stack through the following five internal steps.

Access point matching. The first step is access point matching, which examines packet headers. In cloud environments, services under the same tenant often share a common IP address and port number to simplify external client access. We define this shared identifier as an *access point (AP)*, grouping related services under a unified identifier. As shown in Figure 4, services *a* and *b* share the same AP *i*. The AP detector, implemented on the NIC's programmable pipeline [21], identifies the matched AP for incoming packets by inspecting headers at line rate. Packets that do not match any AP are immediately forwarded to the node's kernel stack. Once an AP is identified, the packets, along with their AP information, are assigned to one of the traffic dispatchers for further steps.

TLS decryption. Ingress gateways can terminate downstream TLS connections with external clients and relays packets through new upstream connections with internal services. Thus, packets received by the NIC stack may contain encrypted payloads, which would otherwise prevent traffic dispatchers from analyzing them. To analyze TLS packets, the NIC stack employs the kTLS offloading technique [58], which is a standard kernel feature [45] widely used in cloud environments [47, 57, 61]. In this model, TLS handshakes with clients are handled by the node's kernel as usual, while TLS (de)encryption tasks are offloaded to the NIC stack. During the TLS handshake process, the kernel installs secrets (e.g., session keys) and TLS connection metadata into the AP detector. The AP detector then leverages this information to identify target TLS packets and decrypt their payloads by using accelerators at line-rate before assigning them to traffic dispatchers, allowing them to analyze TLS packets' payload.

AP-aware RSS. To efficiently distribute packets among traffic dispatchers, the NIC stack employs AP–aware receiver-side scaling (RSS). This mechanism dynamically assigns packets from a given AP to a designated set of cores. For instance, packets from AP *i* are directed to one of the cores in core set *i*, where administrators can predefine core sets for each AP through a dedicated policy. Within a core set, a packet for a new connection is assigned to a core in a round-robin manner, and all subsequent packets of that connection remain bound to the same core to ensure packet affinity. This AP–aware RSS enables *HybridMesh* to allocate cores according to the traffic volume or priority of each AP, while improving cache utilization on NIC cores by preserving packet affinity.

Service matching. A traffic dispatcher receives packets belonging to the same AP from a per-core RSS queue. To determine their destination service, it analyzes packet payloads according to AP–specific ingress policies, as shown in Figure 4. *HybridMesh* is designed to support a wide range of

L7 protocols through a flexible policy model, which allows administrators to specify, for each AP, the protocol in use and the L7 fields that must be inspected. The details of our policy model are provided in Appendix 9.1.

When processing a packet, a traffic dispatcher first retrieves its ingress policy from the policy store and then extracts the fields prescribed by the policy. For microservices based on HTTP, the dispatcher identifies HTTP request packets and extracts fields such as host, path, and user-defined headers. Importantly, this mechanism is not limited to HTTP; it also supports other text-based L7 protocols. For example, the dispatcher can extract designated byte ranges or key–value patterns from UDP payloads, with the policy encoding the precise offsets and field formats required for accurate field identification. This enables the dispatcher to accommodate various protocols commonly employed in service mesh environments.

After identifying the service, the dispatcher selects an endpoint pod from the service's endpoint pool. By default, it uses round-robin selection, while policies can partition pods into subsets and distribute traffic using user-defined fields or weights (e.g., routing requests with the header version=v1 to *subset_a* in Figure 4). Note that dispatchers offload regex-based policy matching to accelerators through per-dispatcher job queues. For each packet, a dispatcher enqueues a regex job carrying connection state and extracted fields; accelerators process jobs in batches and return policy IDs, which the dispatcher uses to finalize service/pod selection.

Metadata embedding. Finally, the traffic dispatcher embeds the analysis results into the packet as metadata, as shown in Figure 4. The metadata occupies 6 bytes and encodes the unique identifiers of the selected service, subset, and policy (see § 3.4 for details). To enable efficient kernel-level access on the node side, the dispatcher inserts this metadata into the Destination MAC (DMAC) field (6 bytes). This placement avoids checksum recalculations that would be required if higher-layer (L3–L7) headers were modified. Moreover, this field is deterministic and therefore easy to validate. Specifically, for external packets containing service request data (e.g., HTTP requests), the DMAC field must correspond to the node's MAC address. Based on these criteria, the dispatcher first verifies the original DMAC field and discards packets with invalid values. Only valid packets are then augmented with metadata and delivered to the node's kernel. This metadata embedding mechanism allows the *HybridMesh* node stack to route packets accurately without incurring the overhead of CPU-intensive traffic analysis.

3.3 *HybridMesh* Tunnel

After NIC stack processing, packets are forwarded to the node kernel. The HMT, running in the kernel space, intercepts them and performs the following two operations.

Relocating metadata. Embedding metadata in the DMAC field avoids checksum recalculation in the NIC stack. However, because L2 headers are stripped in the kernel, the node stack running in the user space cannot access this information. To resolve this, the HMT relocates metadata before packets are processed by the kernel stack, as shown in Figure 5. Upon reception, the HMT checks whether the L2 header contains metadata by inspecting the DMAC field. Packets without metadata (the DMAC field is set to the node's MAC address) are forwarded to the kernel stack for normal protocol processing. Otherwise, the HMT extracts the metadata and inserts it into the payload at a policy-defined offset specific to each service. For instance, in HTTP, metadata can be placed in the user-agent field or a user-defined field (see Appendix 9.1 for more details). The modified packet is then forwarded to the kernel stack for TCP/IP processing.[1]

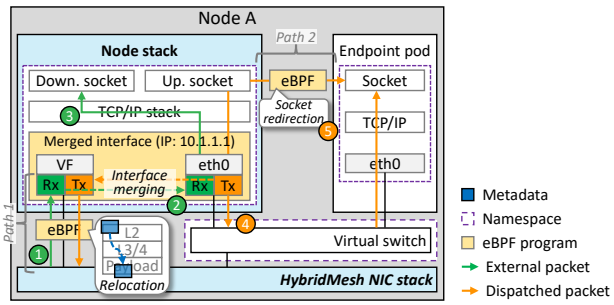

Figure 5: The forwarding path with HMT.

Reducing forwarding path. In a service mesh, the forwarding path of an external packet consists of two segments: (i) the path between the NIC (stack) and the ingress gateway (in our case, the node stack), and (ii) the path between the gateway and the endpoint pod. To optimize these two paths, the HMT employs two different techniques: (i) *interface merging* and (ii) *socket redirection*. For the first path, the HMT leverages SR-IOV technology [1] with our interface merging technique. As shown in Figure 5, the HMT deploys an additional SR-IOV virtual function (VF) in the node namespace. Unlike the primary interface (`eth0`), which remains connected to the virtual switch, the VF directly connects to the NIC stack. Simply adding a new interface would require a separate IP and reconfiguration, but the VF is left without an IP and managed together with `eth0` by the HMT, forming a single logical interface. As shown in Figure 5, ① packets arriving from the NIC stack are ingressed into the node stack's namespace through the VF. ② The HMT intercepts these packets and transfers those carrying metadata from the VF's Rx queue into the Rx queue of `eth0`. Through this mechanism, the two interfaces operate as a single interface under one IP address, allowing the node stack to receive packets from the NIC stack via `eth0`

[1] Note that modifying packet payloads may affect the kernel's checksum verification and lead to packet drops. In our design, however, RX checksumming is offloaded to the NIC hardware. This mechanism, widely adopted in cloud environments as a default setting, ensures that the kernel bypasses checksum verification, since packets with invalid checksums are already discarded by the NIC. Consequently, our metadata relocation does not result in packet drops within the kernel space.

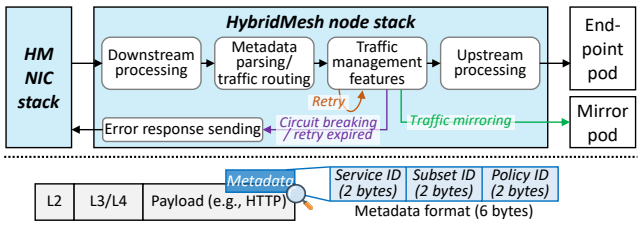

Figure 6: The node stack's workflow with our metadata.

and ③ perform downstream socket processing. Similarly, packets destined for remote pods are redirected by the HMT from `eth0`'s Tx queue to the VF's Tx queue. Consequently, the node stack can exchange external packets directly with the NIC stack via the VF, bypassing the virtual switch, while preserving connectivity with the virtual switch via `eth0`.

For the forwarding path between the node stack and endpoint pods (the second path), the HMT adopts a socket redirection mechanism. ④ When the node stack establishes an upstream socket to an endpoint pod by exchanging handshake packets, the HMT captures the socket descriptors from both ends of the connection. ⑤ As data is transmitted over this socket, HMT intercepts it and directly redirects the data to the corresponding socket of the endpoint pod. This redirection occurs before the data is packetized, eliminating the involvement of the network stack and the virtual switch in the data transfer process. Note that when endpoint pods are located on remote nodes, the dispatched packets are returned back to the NIC stack for physical transmission via the merged eth0 and VF interfaces without going through the virtual switch.

3.4 *HybridMesh* Node Stack

After processing by the HMT, packets are forwarded to the HM node stack. Unlike conventional ingress gateways that must execute CPU-intensive traffic analysis, the node stack utilizes the analysis results produced by the NIC stack.

Metadata-based ingress processing. A naïve design would transfer analysis results from hardware through a dedicated channel between the NIC and node stacks, incurring per-packet overhead. Instead, *HybridMesh* embeds the results directly into each packet as metadata, avoiding extra communication and enabling lightweight access at the node side. Figure 6 shows the node stack's workflow and the metadata format. Upon receiving a packet, the node stack first handles downstream processing, such as establishing TCP/IP connections and managing retransmissions. For data packets containing service requests (e.g., HTTP requests), it extracts the embedded metadata and interprets it. The metadata is 6 bytes long and consists of three fields: service ID, subset ID, and policy ID. The service ID identifies the target service (e.g., a Kubernetes service), while the subset ID indicates the specific pod group within the service. If either field is marked "out-of-policy," meaning the NIC stack could not determine a valid service or subset, the node stack falls back to software handling, similar to existing ingress gateways.

Using these fields, the node stack selects an endpoint pod and retrieves its network information (e.g., IP and port). The policy ID specifies the ingress policy to apply, such as circuit breaking, retries, or mirroring. Finally, the node stack performs upstream socket processing to establish a connection with the chosen endpoint and enforces the policy. For instance, if a pod exceeds its connection limit, the node stack drops the request and returns an HTTP 5xx error (purple arrow in Figure 6). If a pod fails to respond, the request is retried a predefined number of times (brown arrow). It can also duplicate specific requests to a designated mirroring pod (green arrow). This metadata-based processing allows the node stack to focus on traffic management and pod interactions that require control-intensive processing while CPU-intensive traffic analysis can be offloaded to the NIC stack.

Ingress policy updating. The node stack also plays a crucial role in maintaining the freshness of offloaded policies within the NIC stack, adjusting dynamically to changes within the service mesh. These ingress policies comprise two main components: service signatures and endpoint pools. Service signatures include information about active services such as Kubernetes labels, hostnames, HTTP URLs, and ports. In contrast, an endpoint pool is assigned to a service and details the IP addresses, ports, load balancing weights, and subset information for each pod within that service.

To keep these policies up-to-date, the node stack monitors the service mesh's control plane, tracking events related to ingress policies. Upon detecting updates or encountering errors in packet metadata (out-of-policy), it compiles new policies and communicates with the NIC stack's controller to update the policies. The controller and node stack maintain a persistent DMA connection that supports control APIs such as policy-push and state-pull. Through these APIs, the node stack transmits updated policies and retrieves the internal state of the NIC stack (e.g., analysis error counts). Upon receiving updates, the NIC stack controller reconfigures relevant components to reflect the changes.

4 Implementation

***HybridMesh* NIC stack.** We implement the HM NIC stack utilizing the Bluefield-2 SmartNIC [3]. The two modules of this stack are implemented as a set of DPDK applications, running on the ARM cores of the SmartNIC. The traffic dispatcher has a main thread along with seven packet processing threads, each affixed to individual cores. These threads handle incoming HTTP request packets received from the eSwitch via RSS. These traffic dispatchers focus on executing packet pre-processing tasks and offloading regex and crypto operations to a dedicated accelerator via the DOCA core API. The policy manager, pinned to a dedicated core, facilitates DMA communication [7] with the node stack for policy updates and configures other NIC components. In the current version, *Hy-*

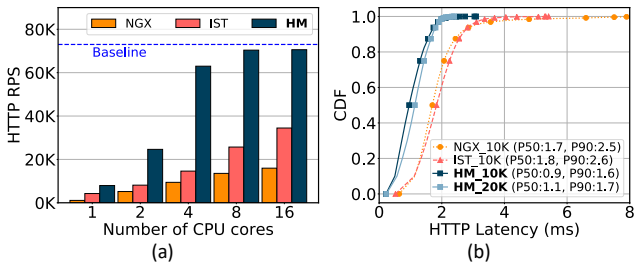

Figure 7: End-to-end ingress processing performance.

bridMesh employs kTLS offloading [58] with *TLS_HW_mode*. The NIC stack handles symmetric key (de)encryption tasks, while other TLS operations remain on the node kernel. TLS metadata and keys required for decryption are offloaded to the NIC stack by the device driver after completing the TLS handshake. The eSwitch identifies packets belonging to target TLS connections and inserts RSS marks. These marked packets are decrypted by the crypto accelerator on the Bluefield-2 [48]. TLS packets that cannot be processed by the NIC are forwarded to the kernel and processed in software.

***HybridMesh* tunnel.** The HMT leverages eBPF [66], comprising three eBPF program types: XDP, TC, and SK_MSG [19]. The XDP program is attached to the node stack's VF and host-side interfaces linked to the virtual switch. It performs metadata reordering when receiving packets from the NIC stack. The TC program is attached to both the original interface and VF of the node stack, implementing our interface merging mechanism. Lastly, the SK_MSG program redirects packets between the node stack and each endpoint pod. Importantly, all these components run in the kernel space, ensuring full compatibility with user space applications.

***HybridMesh* node stack.** We implement the HM node stack by leveraging HAProxy [8], selected for its wide adoption and lightweight design. We extend HAProxy's packet processing pipeline to accommodate the necessary functionalities for our metadata-based ingress processing. We develop a set of custom HTTP filters that, post-downstream processing, analyze HTTP request packets to determine the appropriate backend servers based on metadata. In addition, these filters provide three distinct traffic management features, including circuit-breaking, weighted load balancing, and mirroring.

5 Evaluation

Here, we evaluate the effectiveness of *HybridMesh* by answering the following four questions. (1) How much performance improvement in service mesh ingress processing can *HybridMesh* provide? [§5.2 and §5.3] (2) Can *HybridMesh* maintain scalability with a high number of ingress policies and a high density of connections? [§5.4] (3) Can *HybridMesh* offer better cost/power efficiency than software-only ingress gateways? [§5.5] (4) Can *HybridMesh* support various traffic management features required for service meshes? [§5.6]

5.1 Evaluation Setup

We create a testbed comprising two physical machines (i.e., a client and a server), each equipped with two Intel Xeon Silver CPUs (20 virtual cores each) with 64GB of RAM. The client has a ConnectX-5 NIC, and the server is equipped with a BlueField-2 SmartNIC with the *HybridMesh* prototype. The two machines are directly connected via a 40 GbE cable. On the server side, we install the Istio service mesh [12] atop Kubernetes [18]. Within this mesh, we deploy one Kubernetes service [4], comprising eight endpoint pods. Each pod utilizes one CPU core and runs a simple web server. On the client side, we utilize the *wrk2* tool to generate HTTP traffic for evaluations. The ingress gateway is configured to route traffic from the client machine to the appropriate pods based on port numbers and HTTP headers. To isolate gateway performance from sidecar overhead, we replace the default sidecars with lightweight in-kernel alternatives [6, 10, 62]. These kernel sidecars perform only L3/L4 processing (e.g., NAT and connection tracking) without user-space proxy execution.

5.2 End-to-end System Performance

We first evaluate the performance of *HybridMesh* in comparison to two widely used ingress gateways: Istio Ingress [11] and Nginx Ingress [14]. Performance is then benchmarked against a baseline scenario in which clients communicate directly with endpoint pods via static IP addresses, bypassing ingress gateways entirely. To reflect typical cloud environments where hardware resources are shared among tenant workloads and a provider's system component, we limit the CPU usage of ingress gateways to a range of 1 to 16 cores.

Figure 7 (a) shows the HTTP throughput (requests per second, RPS) as the number of CPU cores assigned to each gateway increases. *HybridMesh* achieves about 85% of the baseline performance (63K RPS) using only 4 cores and reaches 96.6% with more than 5 cores, demonstrating up to a 4.4x improvement over other gateways. In contrast, Istio and Nginx achieve less than 42% of the baseline performance even with 16 cores, equivalent to 4/5 cores of a single CPU socket in our testbed. Notably, this performance level is comparable to that of *HybridMesh* with just 2 CPU cores. In terms of latency, *HybridMesh* consistently outperforms other gateways. Figure 7 (b) presents the CDF of HTTP latency under a load of 10K RPS, where ingress gateways utilize 2 CPU cores, reflecting their default configuration [42]. *HybridMesh* achieves a median (P50) latency that is twice as fast and a tail (P90) latency that is 1.6x faster than other gateways. The performance advantage of *HybridMesh* becomes even more pronounced under higher traffic loads (20K RPS). While Istio and Nginx fail to sustain this load with 2 cores, *HybridMesh* maintains stable performance, achieving median and tail latencies of 1.1ms and 1.7ms, respectively (HM_20K). These evaluation results clearly show that *HybridMesh*'s hybrid de-

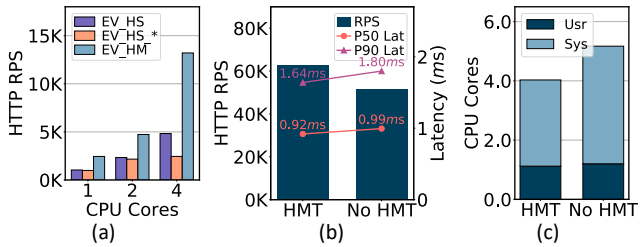

Figure 8: Micro-benchmark of *HybridMesh*: comparison with Hyperscan (a) and the effectiveness of the HMT (b, c).

sign achieves higher performance while requiring fewer CPU resources than existing software-only gateways.

5.3 Micro-benchmark

Traffic analysis acceleration. CPU-level optimization techniques such as Hyperscan [68] can accelerate regex matching. Although we intend to compare Hyperscan-enabled ingress gateways with *HybridMesh* in an end-to-end manner, no existing gateway supports Hyperscan-based matching. To approximate this scenario, we leverage the Intel-managed version of Envoy proxy [5], which provides experimental Hyperscan integration. In our setup, client requests are processed by Envoy, which applies 1,000 routing policies based on HTTP path and host fields and forwards the traffic to the appropriate Nginx pods. We evaluate two configurations: (1) *EV_HS*, which uses Hyperscan to match HTTP headers, and (2) *EV_HM*, which relies on metadata generated by the *HybridMesh* NIC stack. Although the Envoy version used here may exhibit experimental performance, it suffices to highlight the contrast between CPU-based and hardware-offloaded processing.

Figure 8 (a) shows the performance of each Envoy proxy. *EV_HM*, which offloads traffic analysis to hardware, achieves more than twice the throughput of *EV_HS* when using the same number of CPU cores. This performance gap arises from the inherent characteristics of traffic analysis in ingress gateways. Specifically, although Hyperscan improves the efficiency of matching long strings, the L7 fields typically used in ingress gateways [68] are short, rendering Hyperscan's optimization less effective. In contrast, the SmartNIC accelerator performs parallel matching efficiently regardless of string length. Also, we observe that the presence of multiple wildcard (.*) patterns degrades the performance by up to 50% (*EV_HS_**), due to increased search space. In contrast, *EV_HM* maintains consistent throughput regardless of pattern complexity. Although further optimizations may enable Hyperscan's integration with ingress gateways, they cannot address the fundamental limitation of CPU-only processing.

Forwarding path reduction. Next, we evaluate the effectiveness of HMT in reducing the packet forwarding path between the NIC stack and endpoint pods. Figure 8(b) shows the HTTP throughput and latency of *HybridMesh* with and without HMT. With HMT enabled, *HybridMesh* achieves approximately 20% higher throughput and 10% lower latency under a 20K RPS workload. These gains stem from interface merging and socket redirection. Interface merging eliminates one kernel stack traversal, one virtual switch processing step, and one namespace crossing when packets are received at the node stack. Socket redirection further reduces two kernel stack traversals and one virtual switch processing step when forwarding packets to the endpoint pod. Consistent with this behavior, Figure 8 (c) shows that HMT lowers kernel-space CPU utilization (the *sys* field) by about 25%, by bypassing kernel stack operations and virtual switch processing.

5.4 System Scalability

Here, we evaluate the scalability of *HybridMesh* and Istio under varying resource allocation conditions, focusing on two distinct scenarios. In the first scenario, each solution is limited to 2 CPU cores, simulating a default environment where tenant workloads can utilize the majority of CPU cores (HM_C2 and IST_C2). To show the performance advantage of our hybrid design, the second scenario allows software-only ingress gateways to utilize 20 CPU cores (all cores within a single CPU socket), while *HybridMesh* operates with just 4 cores. For clarity, we omit the results of Nginx Ingress, as its performance trends closely resemble those of Istio Ingress.

Connections. Figure 9 (a) shows the HTTP mean latency as the number of concurrent connections increases. Istio exhibits poor scalability under increasing connection loads in both scenarios. When using 2 cores, Istio's latency rises to 75.39ms at 1K connections, and even with 20 cores, it still suffers from high latency, reaching 28ms at 1K connections. These results show Istio's inefficiency in handling heavy connection loads, even with substantial CPU resources. In contrast, *HybridMesh* consistently outperforms Istio across all tested conditions. With 2 cores, *HybridMesh* maintains a low latency of 3.26ms at 1K connections. When using 4 cores, its latency remains under 2ms even at 1K connections, showcasing robust scalability under high connection loads.

Ingress policies. Figure 9 (b) shows the HTTP throughput of each gateway as the number of policies increases. Istio exhibits limited performance with 2 cores, achieving less than 14K RPS with 128 policies and dropping to 5.6K RPS at 1K policies. When using 20 cores, Istio manages 18K RPS under 1K policies, but this remains significantly lower than *HybridMesh*'s performance. By contrast, *HybridMesh* delivers consistently high throughput regardless of the number of policies. With 2 cores, it achieves 25.3K RPS under 1K policies, outperforming Istio by 4.4x. When using 4 cores, *HybridMesh* achieves 60K RPS, surpassing Istio by 3.3x even when Istio uses 5x more (20) CPU cores. Moreover, while Istio suffers significant performance degradation as the number of policies grows, *HybridMesh* maintains stable performance.

The root cause of Istio's scalability stems from its high CPU consumption. Figure 9 (c) shows the CPU usage of

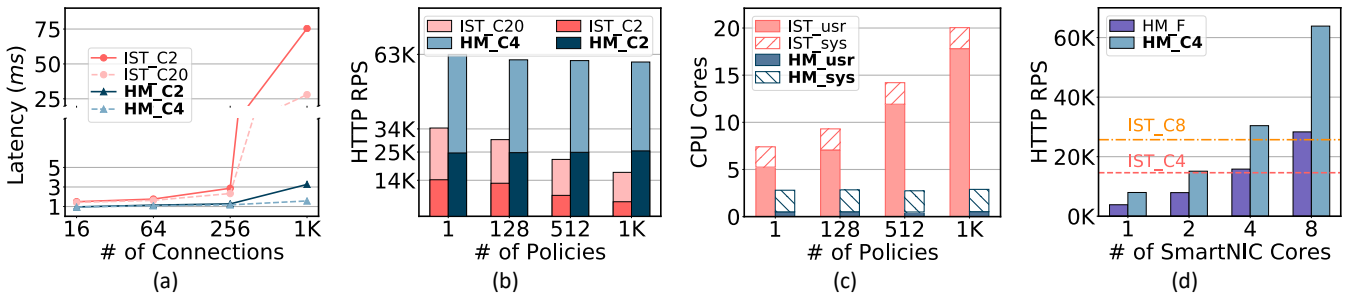

Figure 9: Scalability comparison of *HybridMesh* under different operational conditions: (a) high concurrent connection loads, (b, c) a large number of policies, and (d) varying SmartNIC core allocations.

each solution when handling a 20K RPS traffic load. Istio's user-space CPU usage increases linearly with the number of policies, while kernel-space usage remains constant due to the fixed traffic load. This linear increase results from the growing complexity of policy matching, which demands additional CPU resources and creates bottlenecks that hinder other ingress gateway functionalities. In contrast, *HybridMesh* shows significantly lower CPU utilization, 10x to 34x less than Istio, across all tested configurations. These improvements in performance stem from *HybridMesh*'s hybrid design, which leverages dedicated hardware to minimize the burden on the node CPU, ensuring scalable and efficient ingress processing under high policy and connection loads.

SmartNIC cores. *HybridMesh* relies on SmartNIC cores (S-cores) to operate traffic dispatchers, which manage packet parsing and metadata embedding. Thus, the limited computational capacity of these cores serves as the main performance bottleneck. To assess this impact, we measure HTTP throughput while varying the number of S-cores. For comparison, we also consider a hardware (HW)-only baseline, denoted *HM_F*, in which the entire ingress gateway is fully offloaded to the SmartNIC. In contrast to *HybridMesh*, where the SmartNIC performs traffic analysis and the host CPU handles traffic relaying and management tasks, *HM_F* executes both roles entirely on the S-cores. In this setting, the node stack is migrated onto S-cores and receives packets from the NIC stack via a separate TCP/IP stack that also running on S-cores. After metadata-based matching, the SmartNIC directly dispatches packets to endpoint pods on the CPU-side through per-pod SR-IOV VFs, thereby bypassing the virtual switch.

As shown in Figure 9 (d), both the hybrid and HW-only designs exhibit a linear increase in throughput as the number of S-cores increases. However, when operating with fewer than 2 S-cores, *HybridMesh* does not outperform software (SW)-only solutions (Istio). This limitation arises because, despite offloading regex matching to hardware accelerators, the S-cores still handle critical control logic (e.g., regex job management), creating a bottleneck when only a few S-cores are available. This issue is mitigated with additional S-cores. *HybridMesh* surpasses Istio running with 8 CPU cores when using just 4 S-cores, and the performance gap widens further when all 8 S-cores are used. The HW-only design relies more on S-cores and has lower performance compared to the hybrid design. This disparity occurs because, in addition to executing traffic dispatchers, S-cores must also run the node stack, which includes complex yet non-acceleratable tasks such as upstream/downstream processing and TCP/IP stack operation. This extra workload exacerbates the bottleneck and restricts performance. Nevertheless, the HW-only design surpasses Istio with 8 CPU cores by 2x when utilizing 8 S-cores. Since this design does not consume CPU cores, it is well-suited for resource-constrained environments where CPU cores are primarily allocated to tenant workloads. In summary, the hybrid design provides the most balanced solution by distributing workloads between SmartNICs and CPU cores.

Table 3: Cost effectiveness and power efficiency comparison of different ingress gateway settings under 1K policies.

Hardware setting	*Normalized*			*Cost-effect.*	*Power-efficiency*
	Price	*Power*	*Thr.*		
SW-only [9, 16]	$1,000	100W	1K RPS	1.0	1.0
HW-only [15]	$825	58.5W	0.87K RPS	1.06	1.49
Hybrid (ours)	$921	73.8W	2.1K RPS	**2.3**	**2.87**

5.5 Cost and Power Efficiency Analysis

Beyond its performance advantages, the most compelling benefit of *HybridMesh* lies in its superior cost effectiveness and power efficiency compared to software ingress gateways. These metrics are critical in cloud environments, where they significantly influence the total cost of ownership (TCO). Table 3 presents a detailed comparison of cost effectiveness and power efficiency of the three ingress gateway settings when enforcing 1K policies: SW-only (Istio with 20 cores, all cores in one CPU socket [9]), HW-only design, and hybrid design (*HybridMesh* with 4 CPU cores). For clarity, the price, power, and throughput metrics are normalized to the SW-only design. Note that even without a SmartNIC [15], a standard server requires a conventional NIC for basic network transmission. Therefore, our calculations for the SW-only design include the price and power consumption of a conventional NIC [16],

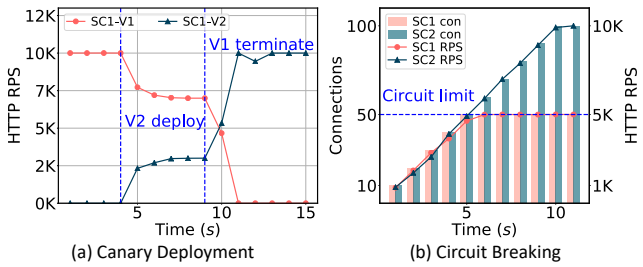

Figure 10: Traffic management features of *HybridMesh*.

which offers the same bandwidth as the SmartNIC used in this paper. Similarly, the hybrid design accounts for the cost and power consumption of 4 CPU cores. Specifically, we compute the cost and power consumption as one-fifth (4 cores) of that in the SW-only design (20 cores) [9]. Cost calculation details for each configuration are provided in Appendix 9.2.

The results show that the HW-only design achieves 1.49x greater power efficiency compared to the SW-only design. Although this design is limited by the computational power of S-cores, the offloading of traffic analysis tasks significantly enhances its power efficiency. The hybrid design further amplifies this advantage, achieving 2.3x higher cost-effectiveness and 2.8x greater power efficiency by optimally distributing workloads between the SmartNIC and CPU cores. These improved cost-effectiveness and power efficiency directly contribute to a lower TCO [40]. In contrast, scaling the SW-only gateway's performance by allocating extra CPU cores proves inefficient, both in terms of cost and power consumption. Hence, dedicating a large number of CPU cores exclusively to an ingress gateway is impractical in cloud environments, as it can impact the performance of tenant workloads [69]. For instance, such configurations introduce contention for CPU cores and the last-level cache, leading to a significant increase in tail latency. Moreover, the widespread adoption of SmartNICs in cloud environments [38,39,46,57,65] further reduces the initial cost of deploying *HybridMesh*, as it is designed to run on commodity SmartNICs (see Appendix 9.3). Thus, our hybrid design emerges as a more suitable solution for service meshes than HW-only or SW-only designs.

5.6 Traffic Management Features

HybridMesh can support various traffic management features required for service meshes. We show this through two examples (see Appendix 9.5 for more examples).

Canary deployment. Figure 10 (a) shows a canary deployment strategy, which is commonly employed for rolling out new versions of a service. Here, a web service (SC1) operates in two versions (V1 and V2), each supported by a distinct subset of endpoint pods. The traffic is incrementally shifted towards the new version (V2). At the 4-second mark, following the deployment of V2, the node stack installs a new policy into the NIC stack to distribute traffic between V1 and V2 at a 7:3 ratio. The NIC stack modifies the load-balancing method for SC1 from round-robin (RR) to weighted RR, allocating traffic to V1 and V2 according to the specified ratio. By the 9-second mark, the node stack modifies the policy to route all traffic exclusively to V2, completely transitioning from V1 to V2. The node stack manages any misrouted packets during policy updates, re-routing them to the correct version (V2) to ensure a smooth service version transition.

Circuit breaking. Figure 10 (b) shows the circuit-breaking feature that prevents pods from being overloaded during sudden spikes in traffic. Here, two services (SC1 and SC2) are deployed, and *HybridMesh* sets a maximum circuit limit of 50 active connections for SC1. As connections increment by 10 every second, generating HTTP requests at 100 RPS, SC2 shows a proportional increase in HTTP throughput. However, SC1's throughput does not surpass 5K RPS due to the node stack's intervention after 5 seconds, which enforces the circuit limit. By leveraging metadata, the node stack directly identifies new request packets for SC1. It then checks the circuit limit for SC1 and responds with errors to any new clients that exceed the limit. Instead of dropping new requests at the NIC stack, the node stack sends proper HTTP responses (e.g., 503), gracefully terminating new requests.

6 Discussion and Limitations

Portability. Although *HybridMesh* is currently implemented on BlueField-2 [3], its design is portable to other SmartNIC classes. For other SoC-based SmartNICs (e.g., Pensando) [22, 26], the HM NIC stack can be reimplemented using vendor SDKs while preserving the same architecture, including the AP detector and traffic dispatchers. The AP detector can be readily realized using programmable match-action pipelines available on these platforms. If the target NIC lacks a hardware regex accelerator, traffic dispatchers can execute entirely on the onboard cores. In this case, they should prioritize simple and frequent patterns (e.g., exact or prefix matching) and offload complex matching to the node stack. Although hardware acceleration is unavailable, performance can be improved using optimized string-matching libraries for the onboard cores, such as VectorScan [28] for ARM. As shown in Figure 11, integrating VectorScan improves throughput by 4.38× over a baseline implementation and achieves over 10Gbps with eight ARM cores. While slower than hardware-accelerated matching, this design still reduces CPU utilization and preserves a fast path for common patterns. For FPGA-based NICs [23, 25], the AP detector and simplified traffic dispatchers can be integrated directly into the FPGA pipeline using existing toolchains [24, 27, 30], outperforming SoC-based SmartNICs. More complex matching over L7 fields should be handled in cooperation with the node stack. Additional discussion is provided in Appendix 9.3.

Protocol support. The current prototype of *HybridMesh* targets HTTP REST APIs due to their widespread adoption [37, 59]. However, it can be extended to support HTTP/2-

Table 4: Comparison of SmartNIC-based systems for service meshes: ● (support), ◖ (partially support), ○ (not support)

System	*Design*	*Traffic dispatching*	*Path opt*	*Traffic mgt*	*Trans-parency*
Speedo [33]	HW-only	L3 – L4	○	○	●
FlatProxy [50]	HW-only	L3 – L7	●	◖	○
QingNiao [67]	Hybrid	L3 – L7	○	●	○
HybridMesh	Hybrid	L3 – L7	●	●	●

based protocols, such as gRPC and QUIC, by incorporating a framer into the NIC-side traffic dispatcher. This framer reconstructs HTTP/2 streams, extracts routing keys from compressed headers, and identifies request boundaries under connection multiplexing prior to policy-based field extraction. Header decompression can be offloaded to built-in accelerators. Fully decoding HTTP/2 headers and maintaining complete per-stream state, however, may be impractical on the onboard cores. Therefore, the framer selectively decodes a small set of high-value routing keys, trading policy expressiveness for efficiency. Further discussion is provided in Appendix 9.4.

NIC resource constraints. Similar to prior SmartNIC-based systems [33,50,56,70,71], *HybridMesh* is ultimately bounded by the resources available on the SmartNIC. In particular, the onboard ARM cores are responsible for L7 header parsing and regex job management, and can therefore become the first bottleneck under high connection churn or traffic bursts, even when hardware accelerators remain underutilized [49]. Moreover, when packets must frequently traverse between the NIC pipeline and the ARM cores, the internal PCIe path can also become a limiting factor. To mitigate these constraints, *HybridMesh* can extend its hybrid design with an explicit CPU-side fallback. Specifically, the NIC stack monitors the load of the onboard cores and, upon overload, forwards a subset of packets directly to the node stack after L3/L4 matching (i.e., AP matching). The node stack then performs software matching only for these packets, leveraging CPU-side optimizations (e.g., Hyperscan) and, when available, invoking NIC accelerators through DMA. This approach keeps accelerators utilized and avoids saturating the onboard cores.

7 Related Work

Software optimization. Service meshes are integral to microservice architectures, prompting numerous studies [36,51,72] that dissect their performance characteristics and identify potential bottlenecks. To address this problem, most existing studies focus on enhancing sidecar performance through software optimizations, utilizing recent advancements in kernel networking features [6,10,64] and shared memory-based data exchanges [31,63]. For instance, Cilium mesh [6] and Istio ambient [10] introduce in-kernel, lightweight alternatives to sidecars, thereby creating a sidecar-less service mesh data plane where inter-service communication is managed by eBPF programs. User-space proxies are engaged only for traffic requiring L7 processing. While these studies enhance inter-service communication performance, they continue to face challenges related to resource overheads and limited performance due to their software-only design. In contrast, *HybridMesh* focuses on enhancing the performance of the ingress gateway through hardware acceleration.

Hardware offloading. The advent of SmartNICs has inspired various systems aimed at enhancing microservice performance [33,50,60,67,70,71]. Speedo [33] improves service invocation latency by offloading traffic dispatching and orchestration tasks to the SmartNIC. QingNiao [67] migrates L7 routing logic from software proxies to a SmartNIC through application–protocol co-design, enabling efficient request distribution across processes. FlatProxy [50] removes software sidecars and implements hardware sidecars on the SmartNIC to accelerate inter-pod communication. However, these systems generally do not fully support the traffic management semantics and routing models required by service meshes, or they require modifications to existing applications, as summarized in Table 4. For instance, Speedo relies on L3/L4 dispatching mechanisms, such as port-based load balancing, and cannot interpret L7 request context (e.g., HTTP headers). FlatProxy focuses on sidecar acceleration and does not provide comprehensive traffic management features; it also requires kernel modifications to enable direct forwarding between pods and offloaded sidecars. Although QingNiao supports accelerated L7 routing via a hybrid design, it depends on application modifications, requiring customized L7 protocols. Moreover, it primarily targets process-level request distribution, making it less suitable for containerized service mesh environments. In contrast, *HybridMesh* is designed to accelerate ingress gateways in service meshes and operates without requiring modifications to applications or protocols.

8 Conclusion

This paper identifies an ingress gateway as a critical bottleneck in the service mesh data plane and presents *HybridMesh*, a novel HW-SW hybrid ingress gateway designed to overcome this limitation. It leverages a SmartNIC to accelerate CPU-intensive traffic analysis while employing a lightweight CPU-side proxy to perform traffic management. Evaluations show that *HybridMesh* achieves 4.4× higher throughput and 2.3× greater cost-effectiveness than SW ingress gateways.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00212738) and the IITP (Institute of Information & Communications Technology Planning & Evaluation) - ITRC (Information Technology Research Center) grant funded by the Korea government (Ministry of Science and ICT) (IITP-2026-RS-2023-00258649).

References

[1] PCI-SIG Single Root I/O Virtualization (SR-IOV) Support in Intel® Virtualization Technology for Connectivity. https://www.intel.com/content/www/us/en/pci-express/pci-sig-single-root-io-virtualization-support-in-virtualization-technology-for-connectivity-paper.html, 2008.

[2] Cncf annual survey 2022. https://www.cncf.io/reports/cncf-annual-survey-2022/, 2022.

[3] Nvidia bluefield-2 dpu. https://www.nvidia.com/en-us/networking/products/data-processing-unit/, 2022.

[4] Service | Kubernetes. https://kubernetes.io/docs/concepts/services-networking/service/, 2023.

[5] Service mesh - envoy regular expression matching acceleration with hyperscan user guide. https://www.intel.com/content/www/us/en/content-details/761773/service-mesh-envoy-regular-expression-matching-acceleration-with-hyperscan-user-guide.html, 2023.

[6] Cilium service mesh. https://cilium.io/use-cases/service-mesh/, 2024.

[7] Doca dma api. https://docs.nvidia.com/doca/sdk/doca+dma/index.html, 2024.

[8] The first and most complete ingress controller implementation for haproxy. https://haproxy-ingress.github.io/, 2024.

[9] Intel® xeon® silver 4114 processor. https://www.intel.com/content/www/us/en/products/sku/123550/intel-xeon-silver-4114-processor-13-75m-cache-2-20-ghz/specifications.html, 2024.

[10] Introducing ambient mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/, 2024.

[11] Istio ingress gateways. https://istio.io/latest/docs/tasks/traffic-management/ingress/ingress-control/, 2024.

[12] Istio service mesh. https://istio.io/, 2024.

[13] Kubernetes: pods. https://kubernetes.io/docs/concepts/workloads/pods/, 2024.

[14] Nginx ingress controller. https://docs.nginx.com/nginx-ingress-controller/, 2024.

[15] Nvidia mellanox® bluefield-2 e-series mbf2m516a-cenot 100 gbe dual-port, fhhl. https://www.tes-itsolutions.com/product-page/nvidia-bluefield-2-e-series-mbf2m516a-cenot-100gbe-dual-port-qsfp56-fhhl/, 2024.

[16] Nvidia mellanox® mcx623106an-cdat 100gbe connectx-6 dx card crypto disabled. https://www.tes-itsolutions.com/product-page/refurbished-nvidia-mcx623106an-cdat-100gbe-connectx-6-dx-card-crypto-disabled, 2024.

[17] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2024.

[18] Production-grade container orchestration. https://kubernetes.io/, 2024.

[19] Program type bpf prog type sk msg. https://ebpf-docs.dylanreimerink.nl/linux/program-type/BPF_PROG_TYPE_SK_MSG/, 2024.

[20] Running multiple nginx ingress controllers. https://docs.nginx.com/nginx-ingress-controller/installation/running-multiple-ingress-controllers/, 2024.

[21] Doca documentation v3.2.1 (2025 lts u1): Hardware architecture. https://docs.nvidia.com/doca/sdk/hardware-architecture/index.html, 2025.

[22] Amd pensando dpus. https://www.amd.com/en/products/networking/pensando.html, 2026.

[23] Amd xilinx alveo data center accelerator cards. https://www.amd.com/en/products/accelerators/alveo.html, 2026.

[24] Amd xilinx vitis networking p4 (netp4) / sdnet. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-networking-p4.html, 2026.

[25] Intel fpga ipu platform. https://www.intel.com/content/www/us/en/products/details/programmable/ipu-platform.html, 2026.

[26] Intel infrastructure processing unit (ipu). https://www.intel.com/content/www/us/en/products/network-io/infrastructure-processing-unit.html, 2026.

[27] Intel open fpga stack (ofs). https://github.com/OFS/ofs-community, 2026.

[28] Vectorscan: an open source scanning engine (hyperscan-compatible). https://github.com/VectorCamp/vectorscan, 2026.

[29] Bhardwaj, A., and Krishna, C. R. Virtualization in cloud computing: Moving from hypervisor to containerization—a survey. *Arabian Journal for Science and Engineering 46*, 9 (2021), 8585–8601.

[30] Bifulco, R., and Rétvári, G. A survey on the programmable data plane: Abstractions, architectures, and open problems. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)* (2018), IEEE, pp. 1–7.

[31] Chen, J., Wu, Y., Lin, S., Xu, Y., Kong, X., Anderson, T., Lentz, M., Yang, X., and Zhuo, D. Remote procedure call as a managed system service. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 141–159.

[32] Choi, S., Shahbaz, M., Prabhakar, B., and Rosenblum, M. λ-nic: Interactive serverless compute on smartnics. In *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos* (2019), pp. 151–152.

[33] Daw, N., Bellur, U., and Kulkarni, P. Speedo: Fast dispatch and orchestration of serverless workflows. In *Proceedings of the ACM Symposium on Cloud Computing* (2021), pp. 585–599.

[34] Duarte Maia, J. T., and Figueiredo Correia, F. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs* (2022), pp. 1–12.

[35] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (2018), pp. 51–66.

[36] Ganguli, M., Ranganath, S., Ravisundar, S., Layek, A., Ilangovan, D., and Verplanke, E. Challenges and opportunities in performance benchmarking of service mesh for the edge. In *2021 IEEE international conference on edge computing (EDGE)* (2021), IEEE, pp. 78–85.

[37] Grambow, M., Meusel, L., Wittern, E., and Bermbach, D. Benchmarking Microservice Performance: a Pattern-based Approach. In *Proceedings of the Annual ACM Symposium on Applied Computing* (2020), pp. 232–241.

[38] Grant, S., Yelam, A., Bland, M., and Snoeren, A. C. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication* (2020), pp. 681–693.

[39] Guo, Z., Lin, J., Bai, Y., Kim, D., Swift, M., Akella, A., and Liu, M. Lognic: A high-level performance model for smartnics. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (2023), pp. 916–929.

[40] HYPOLITE, J., SONCHACK, J., HERSHKOP, S., DAUTENHAHN, N., DEHON, A., AND SMITH, J. M. Deepmatch: Practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies* (2020), pp. 336–350.

[41] ISTIO. Istio ingress: Circuit breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/, 2025.

[42] ISTIO. Istio ingress gateway deployment. https://github.com/istio/istio/blob/master/manifests/charts/gateways/istio-ingress/templates/deployment.yaml, 2025.

[43] ISTIO. Istio ingress: Request timeout. https://istio.io/latest/docs/tasks/traffic-management/request-timeouts/, 2025.

[44] KANG, H., LE, M., AND TAO, S. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E)* (2016), IEEE, pp. 202–211.

[45] KERNEL DEVELOPMENT COMMUNITY, T. Kernel tls offload. https://docs.kernel.org/networking/tls-offload.html, 2025.

[46] KFOURY, E. F., CHOUEIRI, S., MAZLOUM, A., ALSABEH, A., GOMEZ, J., AND CRICHIGNO, J. A comprehensive survey on smartnics: Architectures, development models, applications, and research directions. *IEEE Access* (2024).

[47] KIM, D., LEE, S., AND PARK, K. A case for smartnic-accelerated private communication. In *Proceedings of the 4th Asia-Pacific Workshop on Networking* (2020), pp. 30–35.

[48] KIM, T., NG, D. M., GONG, J., KWON, Y., YU, M., AND PARK, K. Rearchitecting the {TCP} stack for {I/O-Offloaded} content delivery. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)* (2023), pp. 275–292.

[49] LEE, S., YOU, M., KIM, S., AND PARK, T. Comprehensive performance analysis of security applications on the bluefield-3 smartnic. *Computer Networks* (2025), 111830.

[50] LI, M., LU, W., LIN, H., WU, J., ZHANG, Y., AND YAN, G. Flatproxy: A dpu-centric service mesh architecture for hyperscale cloud-native application. *arXiv preprint arXiv:2312.01297* (2023).

[51] LI, W., LEMIEUX, Y., GAO, J., ZHAO, Z., AND HAN, Y. Service mesh: Challenges, state of the art, and future research opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)* (2019), IEEE, pp. 122–1225.

[52] LINGUAGLOSSA, L., LANGE, S., PONTARELLI, S., RÉTVÁRI, G., ROSSI, D., ZINNER, T., BIFULCO, R., JARSCHEL, M., AND BIANCHI, G. Survey of performance acceleration techniques for network function virtualization. *Proceedings of the IEEE 107*, 4 (2019), 746–764.

[53] LIU, G., HUANG, B., LIANG, Z., QIN, M., ZHOU, H., AND LI, Z. Microservices: architecture, container, and challenges. In *2020 IEEE 20th international conference on software quality, reliability and security companion (QRS-C)* (2020), IEEE, pp. 629–635.

[54] LIU, M., CUI, T., SCHUH, H., KRISHNAMURTHY, A., PETER, S., AND GUPTA, K. Offloading distributed applications onto smartnics using ipipe. In *Proceedings of the ACM Special Interest Group on Data Communication*. 2019, pp. 318–333.

[55] NETRONOME. Netronome logo agilio cx smartnics. https://netronome.com/agilio-smartnics/, 2025.

[56] NI, Z., WEI, C., WOOD, T., AND CHOI, N. A smartnic-based load balancing and auto scaling framework for middlebox edge server. In *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)* (2021), IEEE, pp. 21–27.

[57] NOVAIS, F. A., AND VERDI, F. L. Unlocking security to the board: An evaluation of smartnic-driven tls acceleration with ktls. In *NOMS 2024-2024 IEEE Network Operations and Management Symposium* (2024), IEEE, pp. 1–9.

[58] NVIDIA. Nvidia tls offload guide. https://docs.nvidia.com/doca/sdk/nvidia+tls+offload+guide/index.html#src-3342442683_NVIDIATLSOffloadGuide-Setup, 2025.

[59] PAN, T., SONG, E., ZUO, Y., ZHANG, S., SONG, Y., ZHAO, J., HOU, W., LU, J., SUN, X., ZHANG, S., ET AL. Hermes: Enhancing layer-7 cloud load balancers with userspace-directed i/o event notification. In *Proceedings of the ACM SIGCOMM 2025 Conference* (2025), pp. 363–380.

[60] PARK, T., YOU, M., CUI, J., JIN, Y., LEE, K., AND SHIN, S. Mecanic: Smartnic to assist urllc processing in multi-access edge computing platforms. In *2022 IEEE 30th International Conference on Network Protocols (ICNP)* (2022), IEEE, pp. 1–12.

[61] PISMENNY, B., LESOKHIN, I., AND LISS, L. Tls offload to network devices-rx offload. *Mellanox, Dec 3* (2017), 5.

[62] QI, S., MONIS, L., ZENG, Z., WANG, I.-C., AND RAMAKRISHNAN, K. Spright: extracting the server from serverless computing! high-performance ebpf-based event-driven, shared-memory processing. In *Proceedings of the ACM SIGCOMM 2022 Conference* (2022), pp. 780–794.

[63] QI, S., MONIS, L., ZENG, Z., WANG, I.-C., AND RAMAKRISHNAN, K. : High-performance ebpf-based event-driven, shared-memory processing for serverless computing. *IEEE/ACM Transactions on Networking* (2024).

[64] SAOKAR, H., DEMETRIOU, S., MAGERKO, N., KONTOROVICH, M., KIRSTEIN, J., LEIBOLD, M., SKARLATOS, D., KHANDELWAL, H., AND TANG, C. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)* (2023), pp. 969–985.

[65] SUBMIT, S. Key smartnics market research statistics. https://smartnicssummit.com/smartnics-facts/, 2023.

[66] VIEIRA, M. A., CASTANHO, M. S., PACÍFICO, R. D., SANTOS, E. R., JÚNIOR, E. P. C., AND VIEIRA, L. F. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR) 53*, 1 (2020), 1–36.

[67] WANG, T., LIN, J., ANTICHI, G., PANDA, A., AND SIVARAMAN, A. Application-defined receive side dispatching on the nic. *arXiv preprint arXiv:2312.04857* (2023).

[68] WANG, X., HONG, Y., CHANG, H., PARK, K., LANGDALE, G., HU, J., AND ZHU, H. Hyperscan: A fast multi-pattern regex matcher for modern {CPUs}. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)* (2019), pp. 631–648.

[69] XIAO, Y., TOOTAGHAJ, D. Z., DHAKAL, A., CAO, L., SHARMA, P., AND KUZMANOVIC, A. Conspirator:{SmartNIC-Aided} control plane for distributed {ML} workloads. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)* (2024), pp. 767–784.

[70] YOU, M., NAM, J., SEO, M., AND SHIN, S. Helios: Hardware-assisted high-performance security extension for cloud networking. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (2023), pp. 486–501.

[71] YOU, M., SEO, M., KIM, J., SHIN, S., AND NAM, J. Hyperion: Hardware-based high-performance and secure system for container networks. *IEEE Transactions on Cloud Computing* (2024).

[72] ZHU, X., SHE, G., XUE, B., ZHANG, Y., ZHANG, Y., ZOU, X. K., DUAN, X., HE, P., KRISHNAMURTHY, A., LENTZ, M., ET AL. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (2023), pp. 142–157.

```
accessPoint:
  cidr: <CIDR>
  protocol: <TCP|UDP>
  port: <uint16>

match:
  - name: <string>
  http:      # HTTP matching
    method:
      exact: <GET|POST|PUT|...>
      regex:  <pattern>
    path:
      exact:  <string>
      regex:  <pattern>
    host:
      exact:  <string>
      regex:  <pattern>
    CustomHeaders:
      - name: <header>
        exact: <string>
        regex: <pattern>

  raw:       # Raw payload matching
    protocol: <TCP|UDP>
    payload:
      - name: <string>
        offset: <uint32>
        length: <uint32>
        match:
          exact:  <hex|string>
          regex:  <pattern>

route:
  service: { name: <string>,  ID: <uint16> }
  subset:  { name: <string>,  ID: <uint16> }

policy:
  name: <string>, ID: <uint16>
  trafficManagement:
    retries:  { try: <uint8>,  timeout: <uint32> }
    circuitBreaker:         { maxConns: <uint32> }
    mirror:{ service: <string>, subset: <string> }
```

Listing 1: *HybridMesh* policy template with access point and protocol-specific matching.

```
relocationRules:
  - name: http-user-agent-inject
    match:
      service:
        name: <string>
        ID: <uint16>
      L4: TCP
      L7: HTTP
    placement:
      http:
        header_name: "User-Agent"
        length: 6

  - name: raw-payload
    match:
      service:
        name: <string>
        ID: <uint16>
      L4: UDP
      L7: RAW
    placement:
      raw:
        offset: 32
        length: 6
```

Listing 2: Example metadata relocation policy in HMT.

9 Appendix

9.1 Policy Model

Service matching. Listing 1 shows the policy model for *HybridMesh*'s traffic dispatcher. Each policy binds traffic to a logical service via an L3/L4 access point and an L7 protocol-specific matcher. The template supports both HTTP rules on structured fields and a raw payload match on byte ranges specified by an offset and a length field. Upon a successful match, the dispatcher annotates the packet with (serviceID, subsetID, policyID) metadata, enabling efficient policy enforcement without re-analysis at the node stack.

Metadata relocating. Listing 2 shows the metadata relocation policy template in HMT. Our policy allows each service to specify distinct rules for embedding metadata within packet payloads. This flexibility extends beyond HTTP to other text-based L7 protocols, such as UDP-based services, thereby ensuring broad applicability across heterogeneous service mesh environments. When selecting fields for metadata placement, the policy prioritizes non-critical L7 fields whose modification does not affect the fundamental semantics of the protocol. Examples include the HTTP `User-Agent` field or service-specific custom headers, where overwriting values does not interfere with normal protocol operation. In scenarios where critical fields (e.g., HTTP method or path) must be repurposed for metadata, the node stack must be explicitly designed to interpret and process these modified values correctly. To preserve system correctness, this requires leveraging service identifiers (service ID, subset ID, and policy ID) encoded in the metadata to disambiguate and restore the intended semantics.

9.2 Cost and Power Analysis

The cost-effectiveness and power-efficiency results in Table 3 are estimated using publicly available specifications rather than direct hardware measurements. Importantly, these values represent only the computation units required for ingress processing, namely CPU cores and SmartNICs (or normal NICs), rather than the cost or power of the entire server platform. In all experiments, hyper-threading is enabled. Thus, we consider the CPU used in our evaluation as comprising 20 cores in total.

For the software-only design, the cost is estimated by summing the official price of the Intel CPU [9] with that of a commodity NIC [16] providing throughput comparable to the

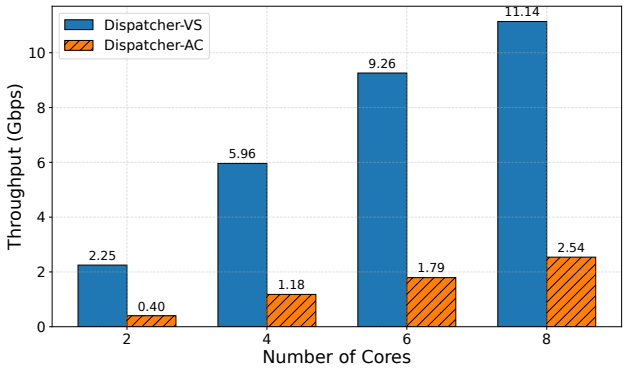

Figure 11: Software dispatcher performance comparison.

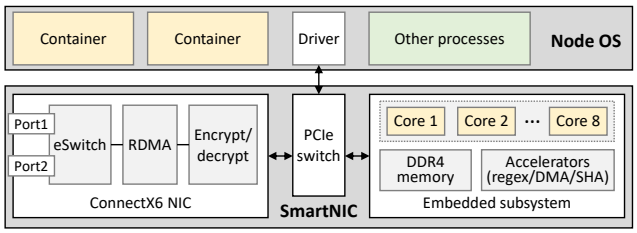

Figure 12: The Bluefield 2 SmartNIC architecture.

SmartNIC. This reflects the necessity of a conventional NIC when the SmartNIC is absent. For the hardware-only design, the cost corresponds solely to the SmartNIC price [15], since all traffic processing is handled on the SmartNIC. For the hybrid design, we include both the SmartNIC price and the cost of a subset of CPU cores (four cores) utilized by the node stack. Specifically, the per-core price is calculated and added to the SmartNIC price.

Power consumption is also derived from vendor specifications. For the SW-only design, we used the average power consumption of the CPU as specified by the vendor for one full socket, which is appropriate since all twenty cores are utilized, and added the power usage of the commodity NIC. For the hybrid design, we combine the SmartNIC's maximum rated power with the estimated consumption of four CPU cores, obtained by dividing the CPU's specified power by twenty and multiplying by four. Because the SmartNIC vendor only publishes maximum power values, this estimate is conservative and likely higher than typical average usage. For the HW-only design, we use only the SmartNIC's maximum rated power. Throughput is measured under the condition of enforcing 1K policies, and all cost, power, and throughput values are normalized against the SW-only design of $1000, 100W, and 1K RPS. Cost-effectiveness is then defined as throughput per unit cost, and power-efficiency as throughput per unit power.

9.3 Portability across different SmartNICs

The current implementation of *HybridMesh* targets BlueField-2, an SoC-based SmartNIC equipped with dedicated processing cores and accelerators, as shown in Figure 12. However, the design concept of *HybridMesh* is not tied to a specific vendor or accelerator. Porting *HybridMesh* to another SmartNIC primarily requires re-implementing the NIC stack using the vendor-provided SDK, while leaving the node stack unchanged.

If the target SmartNIC lacks dedicated regex accelerators, the traffic dispatcher can still execute rule matching on the onboard cores (e.g., ARM) by focusing on simple, fixed matching rules (e.g., exact and prefix matching) and leveraging a core-optimized matching library (e.g., VectorScan). To quantify the benefit of this approach, we implement two non-accelerated dispatchers: one with VectorScan (Dispatcher-VS) and one without it (Dispatcher-AC). We configure 100 exact-match routing rules and generate HTTP traffic using *wrk2*. As shown in Figure 11, Dispatcher-VS improves throughput by 4.38× over Dispatcher-AC, and reaches more than 10 Gbps with 8 ARM cores. While an onboard-core dispatcher does not replace an accelerator-based design, it can still reduce CPU utilization on the node by handling common fast-path rules; requests requiring complex matching are handled by the node stack, while reusing the partial matching results as metadata.

For ASIC-based SmartNICs such as Netronome [55], which lack dedicated regex accelerators, packet classification can instead be executed on many-core network processing units (NPUs). While the absence of accelerators may reduce peak performance, prior studies [40] have demonstrated that many-core NPUs can sustain high throughput and superior energy efficiency for pattern matching tasks. In FPGA-based designs, the NIC stack can be customized through hardware logic or P4-programmable pipelines, where metadata relocation and L7 parsing may be implemented using recirculation-based parsing primitives. Similarly, switch-integrated SmartNICs could adopt RMT pipelines for HTTP parsing and metadata embedding. Overall, *HybridMesh* is not tied to a specific vendor or hardware accelerator. Instead, it provides a general architectural framework that can adapt to diverse SmartNIC platforms, ensuring portability while delivering performance and efficiency benefits tailored to each device's design characteristics.

9.4 Protocol Support

To extend *HybridMesh* to HTTP/2, the NIC stack-side framer must identify individual requests within multiplexed streams and extract routing-relevant metadata with minimal per-stream state. In HTTP/2, each request is associated with a unique *stream identifier* carried in the frame header. Therefore, the framer must parse the fixed 9-byte HTTP/2 frame

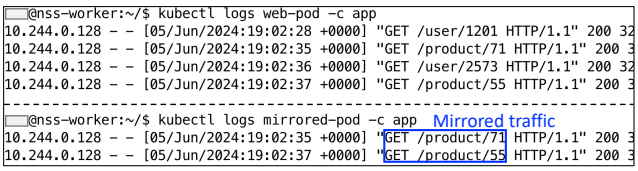

Figure 13: Traffic mirroring of *HybridMesh*; Requests for the *product* API are copied to the mirrored pod.

header, which includes the frame length, type, flags, and stream ID. Request boundaries are determined by tracking HEADERS frames (which initiate a request), optional CONTINUATION frames (for fragmented headers), and DATA frames. The framer maintains lightweight per-stream state consisting of the stream ID, header parsing progress, and an end-of-stream flag derived from frame flags. Full connection-level state reconstruction is unnecessary; instead, the framer operates on a per-stream basis and discards state once a request boundary is identified and the required routing keys are extracted.

For compressed headers, HTTP/2 employs HPACK encoding, which introduces additional complexity. On BlueField-2, header decompression can leverage available on-NIC acceleration primitives to offload Huffman decoding and table lookups. Rather than fully reconstructing all header fields, the framer selectively decodes only a small subset of high-value routing keys (e.g., `:method`, `:path`, and `:authority`). This selective decoding reduces ARM-core overhead and limits dynamic table management to entries relevant for routing. If header compression context becomes too complex or exceeds resource constraints, the framer can fall back to forwarding the request to the node stack for software-based processing. This design preserves efficiency while maintaining compatibility with the HTTP/2 specification and ensuring extensibility within the resource limits of the SmartNIC.

9.5 Traffic Mirroring

In contrast to prior hardware-only studies [32, 33, 54, 60, 70, 71], the hybrid design of *HybridMesh* supports a broader set of traffic management services required for diverse deployment scenarios and effective network debugging in service meshes. The node stack can exploit CPU cores to implement flexible error-handling features that are unsuitable for SmartNIC execution. Beyond the two examples discussed in Section 5.6, we present a third example in Figure 13. This case highlights traffic mirroring, a mechanism commonly employed for microservice debugging. In this setup, HTTP traffic is routed to the *web-pod* endpoint, while traffic matching the URL pattern (/product/d{3}/) is mirrored to a *mirrored-pod*. For packets matching this pattern, the NIC stack assigns a special value (*mirror set*) to the subset ID field in the metadata. Consequently, when the node stack forwards packets to the *web-pod*, it also duplicates those marked with the *mirror set* to the *mirrored-pod*, without incurring the overhead of CPU-intensive traffic analysis.